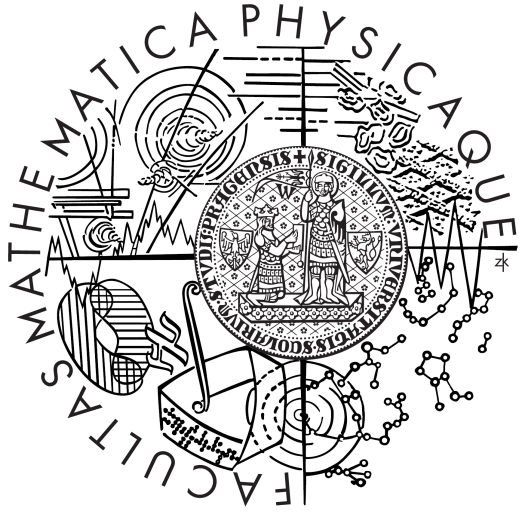
Charles University in Prague
Faculty of Mathematics and Physics

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: Computer Science
Specialization: Theoretical Computer Science

Prague 2013

I would like to express my sincere gratitude to my supervisor RNDr. Petr Pudlák, Ph.D. for the useful comments, remarks and willingness to help me with this thesis. I would also like to thank my parents, who have supported me throughout my studies and whole life.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Typed Functional Genetic Programming

Autor: Tomáš Křen

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: RNDr. Petr Pudlák, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: V této práci je představen design a implementace systému řešícího úlohu genetického programování v simply typed lambda kalkulu. Je zde představena metoda inicializace počáteční populace založená na technice produkující typované lambda termy v dlouhé normální formě. Tato metoda je parametrizována jednoduchou prohledávací strategií. Několik takových prohledávacích strategií je představeno, jako například strategie pro systematické generování nebo strategie odpovídající standardní ramped half-and-half metodě. Další z představených strategií, strategie jménem geometrická strategie je blíže podrobena experimentům, které ukáží že má několik žádoucích efektů na průběh evoluce, jakými jsou zlepšení míry úspěšnosti, nižší časové nároky a menší průměrnou velikost termů v porovnání se standardní ramped half-and-half metodou generování jedinců. Další výkonnostní zlepšení jsou navržena a podpořena experimenty, jedná se o η-normalizaci vygenerovaných jedinců a @-tree reprezentaci jedinců. Použitý proces eliminace abstrakcí umožňuje použití jednoduchého podstromy měnícího křížení.

Klíčová slova: genetické programovaní, funkcionalní programovaní

Title: Typed Functional Genetic Programming

Author: Tomáš Křen

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Petr Pudlák, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In this thesis is presented design and implementation of a system performing genetic programming in simply typed lambda calculus. Population initialization method based on term generating technique producing typed lambda terms in long normal form is introduced. This method is parameterized by simple search strategy. Several search strategies are presented, such as strategy for systematic generation or strategy corresponding to standard ramped half-and-half method. Another such a strategies called *geometric* strategy is further examined in experiments and shown to have various desirable effects such as improved success rate, lesser time consumption and smaller average term size in comparison with standard ramped half-and-half generating method. Other performance enhancements are proposed and supported by experiments such as η-normalization of generated individuals and @-tree representation of individuals. Abstraction elimination is utilized to enable use of simple tree-swapping crossover.

Keywords: genetic programming, functional programming

Contents

Introduction

Genetic programming (GP) is an AI technique, which falls into broader category of evolutionary algorithms — mataheuristic search algorithms inspired by biological evolution by natural selection. I dare say many people perceive its beauty in the fact that GP is a computer program constructing other computer programs with desired properties by breeding them. It is also pretty successful technique in a number of areas [2]. And perhaps that is why GP has become very popular.

GP constructs the population of solution programs from two sets of building blocks; set F of functions and set T of constants and variables. In standard GP *closure* of those building blocks is required. This means that all functions in F must accept all constants and variables from T as their arguments and produce such values that are also acceptable by all functions in F as arguments. Main motivation behind this thesis is to design and implement a system that eliminates this constraint. This can be achieved by utilization of *types*. Utilization of types in GP is not a new idea; it was dealt with in various previous works [4, 5, 6, 7].

This thesis deals with this task using concepts from theory of typed λ-calculus and combinatory logic and shows how to use them in GP. Hopefully, it is also shown that those two fields are nicely linked together. Several enhancements are derived from this theoretical background and shown to have positive effect on performance of our system. Our main focus lies in the process of generating the initial population.

Design of a GP system utilizing these methods is presented. This design aims to be generalization of standard GP developed by Koza[1], rather than being specialized variant of GP using more recent techniques. This decision is based on belief that simplicity of design of standard GP plays nicely with testing those λ-calculus concepts in the evolutionary context and on belief that modern specialized techniques can be incorporated later.

Notable highlights of the thesis are:

- *Inhabitation tree* is presented and used for *generation of individuals*. This generation process is parameterized by simple *search strategy*. Several search strategies are presented, such as *systematic* strategy performing exhaustive enumeration of individuals and strategy performing generalization of standard GP term generating method *ramped half-and-half*.

- One of proposed search strategies, the *geometric* strategy, is examined in a couple of experiments and shown to have several desirable properties.

- The *η-normalization* of generated individuals and *@-tree* representation of individuals, two enhancements of the system inspired from the theoretical background are examined and shown to have improving effect on performance of the system.

- *Abstraction elimination* is utilized to enable use of simple *mutation operator*. Both simple and optimize versions of this transformation are examined.

Described system is implemented in purely functional language *Haskell*. Evaluation of individual λ-terms is performed by translating them to Haskell programs and evaluating them in Haskell interpreter. The core part of the system runs as server, which is managed through user interface accessible via web browser. Solution individuals are also translated to *JavaScript* expressions, which enables an interactive solution presentation. This approach helps us with demonstrating that λ-terms can be easily translated to a widely used programming language.

Several example problems for the designed system are presented and examined. Those examples are not intended to bring breakthrough results; they are intended to serve as demonstration of that the system is working properly, as instrument for examining presented techniques and as demonstration of several directions in which the system can be used.

1. Genetic Programming

Genetic programming 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 in 1992 [1].

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*[1]. It is a 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:

- 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 in greater detail.

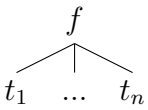
[1]Throughout this thesis, we use the term *building blocks* differently than it is sometimes used in GP context. Hopefully it will not confuse the reader.

1.1 Program trees

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

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

There is straightforward tree representation corresponding to these two cases:

1. One node tree s.
2. 

   where $t_1, ..., t_n$ are trees corresponding to S-expressions $e_1, ..., e_n$.

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

1.2 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* $= T \cup F$.

For each symbol in $T \cup F$ that is not variable there must also be an implementation. 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 must be type A that for every function symbol $f \in F$ the corresponding function implementation standing behind this symbol must be of a type $A \times ... \times A \to A$ and must be total (defined on all combinations of inputs). And analogically for $t \in T$ being of a type A.

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

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

Satisfying this constrain ensures that every program tree build from $T \cup F$ will be total. We will refer to this constrain as to *closure*.

We can say that the motivation behind this thesis is to construct system, where building blocks may be of any type.

1.3 Generating individuals

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

An individual tree is constructed by random (uniform) selection of symbol from subset of $T \cup F$ for the root node and after that by generation of its subtrees recursively. Number of subtrees of a node corresponds to the number of arguments for the selected symbol.

There are two generating sub-methods called *full* and *grow*; each of them restricting differently the subset of $T \cup F$. This restriction depends on depth of the node for which the symbol is being selected.

In order to perform *ramped half-and-half* method one of those two sub-methods is randomly (uniformly) selected and the selected one is performed. This selection is done for each generated tree, so there is 50% chance for tree to be generated by *full* and 50% chance for tree to be generated by *grow*.

A maximum depth $D_{initial}$ is defined (e.g. $D_{initial} = 6$).

For each tree is selected its depth d from $\{2, \ldots, D_{initial}\}$ randomly (uniformly).

In the root ($depth = 0$) both *full* and *grow* select the symbol from the set F.

In nodes with $depth < d$ the *full* method selects from the set F, whereas the *grow* method selects from whole $T \cup F$.

In nodes with $depth = d$ both methods select from the set T.

Generated individual is added into the initial population only if the initial population does not yet contain this individual. This way it is guaranteed that every individual is unique in the initial population.

1.4 Evaluation

Evaluation is process of assigning fitness value to all individuals in the population. It is also usual to check individuals for being correct solution. If such a correct solution is find, the algorithm terminates. In our implementation fitness function returns pair of real number (fitness value) and boolean value (indicating whether tested individual is correct solution).

1.5 Selection

In the field of evolutionary algorithms 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 the *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:

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

1.6 Crossover

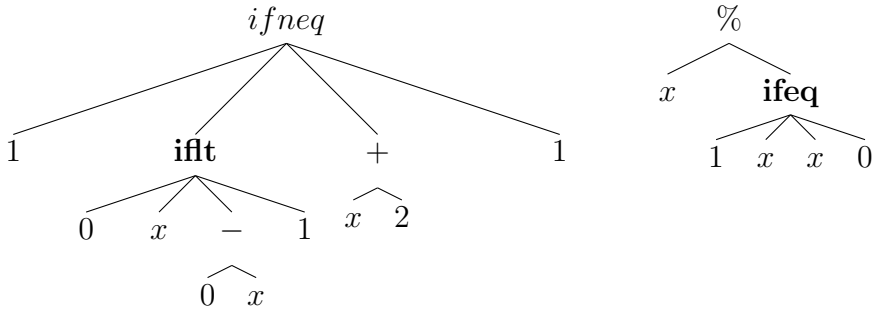
For Koza the most important genetic operator in GP is crossover. It is operator inspired by sexual reproduction occurring in the nature. Generally speaking crossover takes two (parent) individuals and combines theirs genomes to produce two possibly new (offspring) individuals.

In GP the most common crossover is *Subtree swapping crossover*. This crossover randomly selects one node in each parent tree.

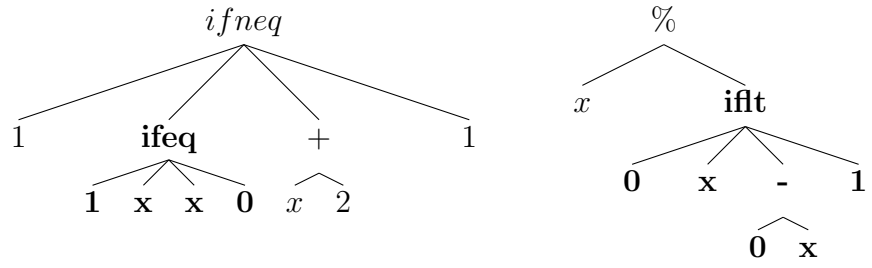
With probability p_{ip} (e.g. $p_{ip} = 90\%$) the crossover node is randomly (uniformly) selected from internal nodes (i.e. function nodes) and with probability $1 - p_{ip}$ (e.g 10%) from leaf nodes. This is done for both parents.

Two new child individuals are constructed by swapping subtrees that 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:



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.

1.7 Reproduction

Reproduction is simple mechanism providing preservation of solutions from the current population to the next one. It simply copies one individual to the next population.

1.8 Mutation

Mutation is genetic operator modifying one individual. There are many options for mutation mechanisms for us to choose from. Here will be described *Subtree generating mutation* witch uses mechanism for generating individuals.

This mutation randomly selects one node in the individual tree. New mutant individual is constructed by replacement of subtree with root in the selected node by new generated tree. This new tree is generated by mechanism for generating individuals.

Analogically as for crossover, a maximum permissible depth for mutant individual is defined.

Similarly as Koza in [1], we do not use the mutation operator in this thesis. This does not mean that I do not like it. Conversely, I am looking forward too its enhancing effect in the future work.

1.9 Construction of next population

Let pop_t be the current population that we want to transform into the next population pop_{t+1}. We start by initializing pop_{t+1} by empty population.

Then we iteratively fill pop_{t+1} by individuals returned by genetic operators.

In each iteration one genetic operator is randomly selected. Then required amount of individuals for the operation is selected by individual selection mechanism described above (one or two individuals for our genetic operators). And those selected individuals are used as input for selected genetic operator which produces some new individuals. Those new individuals are inserted into pop_{t+1}.

This process continues until pop_{t+1} is filled with $popSize$ individuals.

Selection of genetic operation in each operation is controlled by probabilities for each genetic operator.

Koza[1] uses as default probabilities those values:

- *Crossover* : 90%
- *Reproduction* : 10%
- *Mutation* : 0%

1.10 GP algorithm in pseudocode

Bellow is described GP algorithm in pseudocode. It is slight modification of the standard GP algorithm, because the best individual of the generation is preserved unchanged into the next generation.

```
function GP( fitness, T ∪ F, popSize, numGens, probabs )
    gen ← 0
    pop ← generateInitialPopuletion( T ∪ F )
    (popWithFitness, terminate, best) ← evaluate(fitness, pop)

    while gen < numGens ∧ ¬terminate do
        newPop ← empty population
        newPop.insert( best )

        i ← 1
        while i < popSize do
            op ← probabilisticallySelectOperation(probabs)

            switch op do
                case Crossover
                    parent1 ← selection( popWithFitness )
                    parent2 ← selection( popWithFitness )

                    (child1,child2) = crossover( parent1 , parent2 )

                    newPop.insert( child1 )
                    newPop.insert( child2 )
                    i ← i + 2
                case Reproduction
                    indiv ← selection( popWithFitness )
                    newPop.insert( indiv )
                    i ← i + 1
                case Mutation
                    indiv ← selection( popWithFitness )
                    mutant ← mutate( indiv , T ∪ F)
                    newPop.insert( mutant )
                    i ← i + 1

        pop ← newPop
        (popWithFitness, terminate, best) ← evaluate(fitness, pop)
        gen ← gen + 1

    return popWithFitness
```

Let us clarify the input arguments:

- *fitness* - Fitness function.
- $T \cup F$ - Building blocks accompanied with an information about implementations and argument numbers.
- *popSize* - Population size.
- *numGens* - Number of generations.
- *probabs* - Genetic operators probabilities.

Behaviors of contained procedures are hopefully clear from verbal description.

For more detailed information on subject of GP and its many variants see [3] containing many references to other works.

2. Mathematical background

In this chapter we briefly describe key concepts of simply typed λ-calculus. We try to favor readability over formality, for more detailed and formal treatment of this subject see [10, 11].

After introductory description, an analysis of term generation follows; a path from inference rules to term generating grammar rules is shown, *long normal form (lnf)* is described, followed by term generating grammar for *lnf* and analysis of its relevant properties (such as its relation to $\beta\eta$-*nf*), then *inhabitation tree* is presented — an implementation friendly refinement of *lnf* term generating grammar.

2.1 λ-terms

Here we describe programming language, in which we generate individual programs — so called λ-terms.

Definition. 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*.

It is usual to not distinguish between V and C and just use one set of variable names. But because this distinction is quite convenient for us in the implementation, we use it even here. Hopefully, it will not confuse the reader.

We can use $V = \Sigma^+$, set of all non-empty finite strings of symbols from Σ where Σ is some alphabet set, e.g. $\Sigma = \{a, \ldots, z\}$.

And C is set supplied by "the user" to enrich the language with constant names standing for some predefined behavior.

A constant may stand just for another λ-term or it may stand for some predefined constant such as 0,1,2,... and primitive operations on them, e.g. addition. But the specific implementations of these constants will not be ours big concern while reasoning about methods for generating λ-terms.

We use upper case symbols such as $M, N, M_1, M_2, ..., N_1, N_2, ...$ to denote arbitrary lambda terms in contrast with those in lower case such as $a, b, c, ..., x, y, z$, $f_1, f_2, ..., g_1, g_2, ...$ denoting variable names, elements of V.

Sometimes we will use lower case symbols to denote arbitrary variable names (such as x in $(\lambda\, x\,.\, M)$), and at other times, we will use them as specific variable names in specific terms (such as x in $(\lambda\, x\,.\, x)$). The distinction between the two should be clear from the context.

We use bold upper case symbols such as $\mathbf{S}, \mathbf{K}, \mathbf{I}$ as abbreviation for specific fixed terms, e.g. $\mathbf{I}$ always means $(\lambda\, x\,.\, x)$.

We use following notation abbreviations for better readability.

1. $M_1\ M_2\ M_3\ \ldots\ M_n$ for $(\ldots((M_1\ M_2)\ M_3)\ \ldots\ M_n)$
2. $\lambda\, x_1 x_2 \ldots x_n\,.\, M$ for $(\lambda\, x_1\,.\,(\lambda\, x_2\,.\,\ldots(\lambda\, x_n\,.\, M)\ldots))$

For example we can write $(\lambda\, f\,.\,(\lambda\, g\,.\,(\lambda\, x\,.\,((f\ x)\,(g\ x)))))$ as $\lambda\, f\, g\, x\,.\, f\ x\,(g\ x)$.

Here follow some examples of λ-terms.

x
$(\lambda\, x\,.\, x)$
$(\lambda\, x\,.\, f)$
$((\lambda\, x\,.\,(x\ x))\ (\lambda\, x\,.\,(x\ x)))$
$(\lambda\, x\,.\,(\lambda\, y\,.\, x))$
$(\lambda\, f\,.\,(\lambda\, g\,.\,(\lambda\, x\,.\,((f\ x)\ (g\ x)))))$

2.1.1 Subterms, free variables and substitution

In order to demonstrate what is a subterm of a term let us define inductively $\mathrm{ST}(M)$ set of all subterms of a term M.

$$\begin{aligned}\mathrm{ST}(x) &= \{x\}\\ \mathrm{ST}((P\ Q)) &= \{(P\ Q)\} \cup \mathrm{ST}(P) \cup \mathrm{ST}(Q)\\ \mathrm{ST}(\lambda\, x\,.\, P) &= \{\lambda\, x\,.\, P\} \cup \mathrm{ST}(P)\end{aligned}$$

$\mathrm{FV}(M)$ the set of free variables of a term M is defined inductively as follows.

$$\begin{aligned}\mathrm{FV}(x) &= \{\} && \textbf{if } x \in C\\ \mathrm{FV}(x) &= \{x\} && \textbf{if } x \in V\\ \mathrm{FV}((P\ Q)) &= \mathrm{FV}(P) \cup \mathrm{FV}(Q)\\ \mathrm{FV}(\lambda\, x\,.\, P) &= \mathrm{FV}(P) \setminus \{x\}\end{aligned}$$

A variable in λ-term M is called *bound* if it is not free.
λ-term M is called *combinator* if $\mathrm{FV}(M) = \emptyset$.

Notable examples of combinators are combinators $\mathbf{S}$, $\mathbf{K}$ and $\mathbf{I}$.

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

As will be shown in 3.3.1 these combinators will help us in performing crossover of λ-terms.

$M[x := N]$ the substitution of a term N for the free occurrences of a variable x in a term M is defined inductively as follows.

$$\begin{aligned}
x[x := N] &= N \\
y[x := N] &= y && \text{where } x \neq y \\
(P\ Q)[x := N] &= (P[x := N]\ Q[x := N]) \\
(\lambda\, x\,.\, P)[x := N] &= \lambda\, x\,.\, P \\
(\lambda\, y\,.\, P)[x := N] &= \lambda\, y\,.\, (P[x := N]) && \text{where } x \neq y, y \notin \mathrm{FV}(N)
\end{aligned}$$

In order to satisfy the $y \notin \mathrm{FV}(N)$ condition, names of bound variables are chosen such that they differ from the free ones in the term.

2.1.2 β-reduction

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

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

$$(\lambda\, x\,.\, M)N \rightarrow_\beta M[x := N] \tag{2.1}$$

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

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

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

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

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

We read those relations as follows.

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

2.1.3 η-reduction

Similarly as for β-reduction we can define η-reduction except that instead of 2.1 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 $=_{\beta\eta}$.

2.1.4 η^{-1}-reduction

η^{-1}-reduction (also called η-expansion) is the reduction converse to η-reduction. Again it may be obtained by replacing 2.1, now with:

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

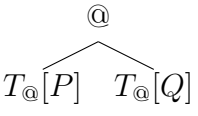
2.1.5 β-normal form

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*.

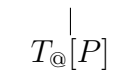
2.1.6 Tree representations of λ-terms

From the definition of λ-term we can straightforwardly derive the standard tree representation for λ-terms, which we call *@-tree*. Term M is translated into @-tree $T_@[M]$ by following rules.

1. $T_@[x] := x$ (i.e. $x \in V \cup C$ becomes one node tree x)
2. $T_@[(P\ Q)] :=$ 

3. $T_@[\lambda x.P] :=$ 

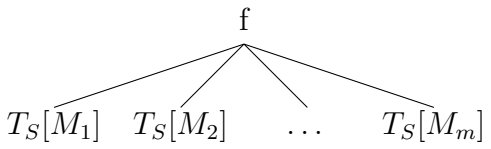
We can enhance this representation by replacing third rule by following one. It compressing consecutive λ-abstractions into one @-tree node.

$T_@[\lambda x_1 \ldots x_n.P] :=$ 

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

Following tree representation is extension of *S-expression trees* described in 1.1, thus we call it *sexpr-tree*. Unlike @-tree, it is not able to cover all λ-terms, but only those in β-nf. Term M is translated into sexpr-tree $T_S[M]$ by following rules.

1. $T_S[x] := x$
2. $T_S[(fM_1M_2 \dots M_m)] :=$



3. $T_S[\lambda x_1 \dots x_n.M] := \lambda x_1 \dots x_n$
   $|$
   $T_S[M]$

2.2 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*). After putting those two pieces (*λ-terms* and *types*) together we will get system called *simply typed λ-calculus*.

Definition. 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.

Usually A is called *set of type variables*, but since it might be source of confusion with polymorphic type variables we prefer this name, which also better suits our usage.

Specific definition of A is not so important, but for practical purposes we can think of it as the set of all non-function types relevant in our situation, e.g., $A = \{Int, IntList, Bool, BoolList, Char, String\}$. Or we can think of A more generally as of set of all non-empty strings of symbols of some alphabet.

We use following notation abbreviation for better readability:
$\tau_1 \to \tau_2 \to \ldots \to \tau_n$ means $\tau_1 \to (\tau_2 \to (\ldots \to (\tau_{n-1} \to \tau_n) \ldots))$.

In other words, the operator $\to$ is right-associative.

It is easy to see that every $\sigma \in \mathbb{T}$ may be expressed as $\tau_1 \to \ldots \to \tau_n \to \alpha$ where $\alpha \in A$ and $n \geq 0$.[1]

Technically speaking, we have types only for functions with one argument, but this property suggests how we can "simulate" function which takes n arguments.

This technique (or trick) is called *Currying*.[2]

2.2.1 Statements of the form $M : \sigma$

Definition. 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 type σ is called the *predicate* and the term M is called the *subject* of the statement $M : \sigma$.

We can look on this relation in tree different ways.

[1] We can prove it by induction on size of σ. For $\sigma = \alpha \in \mathbb{T}$ is $n = 0$ and for $\sigma = \tau_0 \to \rho$ we have from induction hypothesis that $\rho = \tau_1 \to \ldots \to \tau_n \to \alpha$, therefore $\sigma = \tau_0 \to \tau_1 \to \ldots \to \tau_n \to \alpha$.

[2] More formally, currying is process of transforming a function which takes multiple inputs into function that takes one argument and returns function with one less argument.

Programming point of view. Statement $M : \sigma$ means that value M is an instance of data type σ. So statement $M : \sigma$ is somehow analogous to *Java* expression M `instanceof` σ.[3]

Type *constrains* the value and gives *information* about it, e.g., function $f : \sigma \to \tau$ is constrained to only accept values of a type σ as input and must return value of a type τ as output.

Set point of view. Statement $M : \sigma$ can also be viewed as relation $M \in \sigma$. This makes us look on types as on collections of their instances.

Logic point of view. According to the *Curry–Howard correspondence* we can read statement $M : \sigma$ as *"M is a proof of formula σ"*. We will not treat this interesting subject in much detail, let us just say that $\to$ corresponds to the implication $\Rightarrow$ and *atomic types* correspond to *propositional variables*. The whole system of *simply typed λ-calculus* then corresponds to the *implicational fragment of propositional intuitionistic logic*.

Intuition is that the proof of $A \Rightarrow B$ can be imagined as function expecting proof of A and utilizing it in production of the proof of B that is returned as output of the function.

We are mentioning this beautiful correspondence in hope that it will help clarify use of *inference rules*, which are key concept in the process of deriving valid statements about terms and types.

2.2.2 Context

Definition.

1. A *declaration* is a statement $x : \sigma$ where $x \in V \cup C$.
2. A *context* (or *basis*) is set of declarations with distinct variables as subjects.

Again our definition slightly differs from the standard one in that it permits more than one declaration with the same *constant*, which enables us to simulate very primitive kind of ad-hoc polymorphism by *"constant overloading"*. And again it does not affect the theory; it just seems less confusing to me to state these definitions as they are later used in the implementation.

In our system, the set of *building blocks* (the equivalent of the set $T \cup F$ in standard GP) will be a context with all declarations having *constants* as subjects, i.e., a set of constants together with their types.

In the process of term generation, this initial context is being further extend with variables — by "entering inner scopes of bodies of lambda functions".

Notation $\Gamma, x : \sigma$ denotes $\Gamma \cup \{(x : \sigma)\}$ such that Γ does not contain any declaration with x as subject.

[3] This analogy is inadequate in that instanceof is construct of the language, whereas : is symbol out of the language, it enables us to talk about terms from outside perspective.

2.2.3 Simply typed λ-calculus

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

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

Those rules are usually stated in the form of *inference rules*.

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \textit{(axiom)}$$

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

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda\, x\, .\, M) : \sigma \to \tau} \qquad \textit{(}\to\textit{-introduction)}$$

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

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

This statement is derived as follows.

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

If we turn the derivation upside down, we can see that it is a tree with desired statement in the root and *axioms* in the leafs.

New valid inference rules may be produced by finite composition of other valid inference rules.

Let us consider following rule.

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

Its validity is shown by the following rule composition.

$$\dfrac{\dfrac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda\, x\, .\, M) : \sigma \to \tau} \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash ((\lambda\, x\, .\, M)\ N) : \tau}$$

Let us consider following strange rule.

$$\frac{\Gamma, x : \rho \to \pi \vdash M : \tau \qquad \Gamma, y : \rho \vdash N : \pi}{\Gamma \vdash ((\lambda\, x\, .\, M)\ (\lambda\, y\, .\, N)) : \tau}$$

Its validity is shown by the following rule composition.

$$\frac{\dfrac{\Gamma, x : \rho \to \pi \vdash M : \tau}{\Gamma \vdash (\lambda\, x\, .\, M) : (\rho \to \pi) \to \tau} \qquad \dfrac{\Gamma, y : \rho \vdash N : \pi}{\Gamma \vdash (\lambda\, y\, .\, N) : \rho \to \pi}}{\Gamma \vdash ((\lambda\, x\, .\, M)\ (\lambda\, y\, .\, N)) : \tau}$$

Generally speaking, inference rules are used for deriving statements of a form $\Gamma \vdash M : \sigma$ from yet derived statements of such a form (with exception of *axiom* rule, which serves as a "starting point"). Those inference rules are written in the following form.

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \qquad \Gamma_2 \vdash M_2 : \sigma_2 \qquad \cdots \qquad \Gamma_n \vdash M_n : \sigma_n}{\Gamma_{n+1} \vdash \mathbf{T}(M_1, M_2, \ldots, M_n) : \sigma_{n+1}}$$

Where $\mathbf{T}(M_1, M_2, \ldots, M_n)$ stands for some specific λ-term that may contain *"meta-variables"* $M_1, M_2, \ldots, M_n$.

In the example above $\mathbf{T}(M, N) = ((\lambda\, x\, .\, M)\ (\lambda\, y\, .\, N))$

Suppose we have derived statements $\Gamma_1 \vdash M_1 : \sigma_1, \Gamma_2 \vdash M_2 : \sigma_2, \ldots, \Gamma_n \vdash M_n : \sigma_n$. Then we are allowed to use the inference rule to derive statement $\Gamma_{n+1} \vdash \mathbf{T}(M_1, M_2, \ldots, M_n) : \sigma_{n+1}$.

Nice thing about *simply typed lambda calculus* is that program M such that $\Gamma \vdash M : \sigma$ always terminates (provided that implementations of function symbols in Γ also always terminate). This fact is particularly handy for GP — we can evaluate bred programs without worrying about hanging in the infinite loop forever.

2.2.4 Tree representations of typed λ-terms

Simply typed λ-calculus as described above is formally called the *system* $\lambda^{\to}$*-Curry*. Other slightly different formalization is called *system* $\lambda^{\to}$*-Church*. They differ in that $\lambda^{\to}$-Curry uses as terms above described untyped terms, whereas $\lambda^{\to}$-Church uses slightly modified terms with information about types of variables written in lambda abstraction along with variable name. Since those systems are proven to be equivalent, we are not giving this subject much attention here. For more detailed information on this subject see [10].

In our implementation we also use modified representation of λ-terms storing information about types. But our representation is more radical than that of $\lambda^{\to}$-Church; it stores information about types of all subterms.

Regarding tree representation of such a typed λ-term, most of the time it is more convenient to understand this information as invisibly sitting in the tree nodes alongside with visible term symbols.

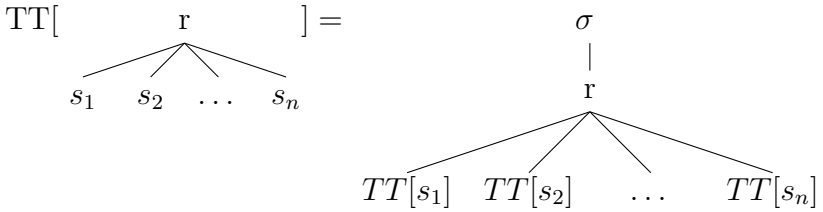
But as we will see below in the section about *inhabitation trees*, there are situations when the explicit type information in the tree representation comes in

handy. Thus we are presenting following tree representation.

Straightforward approach would be to add to each node a visible type entry that would be the type of the subterm corresponding to subtree having this node as the root node.

Approach more suitable for our further purpose is to add a special type node *above* each node.

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



We silently suppose here that type information about every subterm is available.

2.3 Term generating grammar

Inference rules are good for deriving statements of the 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.

Let us start our reasoning about generation of λ-terms with notion of term generating grammar. We will show how to transform an inference rule into a rule of term generating grammar. With the appropriate term generating grammar in hands it is just a simple step to *inhabitation trees*.

It is not a grammar in classical sense, because we will be operating with infinite sets of nonterminal symbols and rules. Technically it it 2-level grammer [11].

Our terminals are symbols used for writing λ-terms.

Our non-terminals are pairs (τ, Γ) where $\tau \in \mathbb{T}$ and Γ is a *context*.

Our initial non-terminal is $S = (\sigma_S, \Gamma_S)$ where σ_S is desired type of generated λ-terms and Γ_S is *building blocks* context.

Let us begin with *axiom* inference rule.

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

For every non-terminal (σ, Γ) and for every x such that $(x : \sigma) \in \Gamma$, there is following grammar rule.

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

Next is $\rightarrow$-*elimination*.

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

For every non-terminal (τ, Γ) and for every type $\sigma \in \mathbb{T}$ there is following grammar rule.[4]

$$(\tau, \Gamma) \longmapsto \Big((\sigma \rightarrow \tau, \Gamma) \_ (\sigma, \Gamma) \Big)$$

And finally $\rightarrow$-*introduction*.

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

For every non-terminal $(\sigma \rightarrow \tau, \Gamma)$ and for every variable x such that Γ does not contain any declaration with x as subject there is following grammar rule.

$$(\sigma \rightarrow \tau, \Gamma) \longmapsto \Big(\ \lambda\ \mathrm{x}\ .\ \ (\tau; \Gamma, x : \sigma) \quad \Big)$$

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 $(\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$ with $\Gamma = \{\}$.

$$\begin{aligned}
&((\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau), \{\}) \\
\longmapsto &\Big(\ \lambda \mathrm{f}.(\sigma \rightarrow \tau, \{(f, \sigma \rightarrow \tau)\}) \ \Big) \\
\longmapsto &\Big(\ \lambda \mathrm{f}.\ \Big(\ \lambda \mathrm{x}.\ \ (\tau, \{(f, \sigma \rightarrow \tau), (x, \sigma)\}) \ \Big) \ \Big) \\
\longmapsto &\Big(\ \lambda \mathrm{f}.\ \Big(\ \lambda \mathrm{x}.\ \ \Big((\sigma \rightarrow \tau, \{(f, \sigma \rightarrow \tau), (x, \sigma)\}) _ (\sigma, \{(f, \sigma \rightarrow \tau), (x, \sigma)\}) \Big) \ \Big) \ \Big) \\
\longmapsto &\Big(\ \lambda \mathrm{f}.\ \Big(\ \lambda \mathrm{x}.\ \ \Big(\mathrm{f} _ (\sigma, \{(f, \sigma \rightarrow \tau), (x, \sigma)\}) \Big) \ \Big) \ \Big) \\
\longmapsto &\Big(\ \lambda \mathrm{f}.\ \Big(\ \lambda \mathrm{x}.\ \ \Big(\mathrm{f} _ \mathrm{x} \Big) \ \Big) \ \Big)
\end{aligned}$$

Let us consider a general inference rule again.

$$\frac{\Gamma_1 \vdash M_1 : \sigma_1 \qquad \Gamma_2 \vdash M_2 : \sigma_2 \qquad \cdots \qquad \Gamma_n \vdash M_n : \sigma_n}{\Gamma_{n+1} \vdash \mathbf{T}(M_1, M_2, \ldots, M_n) : \sigma_{n+1}}$$

The corresponding grammar rule can be summarized as following schema.

$$(\sigma_{n+1}, \Gamma_{n+1}) \longmapsto\ \mathbf{T}\Big((\sigma_1, \Gamma_1)\ , (\sigma_2, \Gamma_2)\ ,\ \ldots\ , (\sigma_n, \Gamma_n) \Big)$$

[4] Terminal symbols for parenthesis and normally *space* now $\_$ (for *function application* operator) are visually highlighted.

2.4 Long normal form

Definition. 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 [11], *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 $L \twoheadrightarrow_\eta B$.*

2.5 Grammar producing λ-terms in *lnf*

In [11] is shown term generating grammar with following rules (our notation is used, but we will not highlighted terminals anymore).[5]

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

The second rule can be replaced by more effective one.

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

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

Following proposition about this grammar is stated (but not proven) in [11]. Since it is key proposition for our term generating method, we also provide a proof.

Proposition 3. *Let context Γ, λ-term M and type σ be given.*
Let $\rightsquigarrow$ be transitive reflexive closure of $\longmapsto$. Then

$$(\sigma, \Gamma) \rightsquigarrow M \Leftrightarrow \Gamma \vdash M : \sigma \text{ and } M \text{ is in lnf.}$$

[5] I was using term generating grammars for term generation on my own before I encountered with [11], where i happily discovered that Barendregt is using almost identical notation. But his *lnf* grammar was more clever then my grammar that was using grammar rules corresponding to the three basic inference rules.

Proof.

Let us start with proving
$(\sigma, \Gamma) \rightsquigarrow M \Rightarrow \Gamma \vdash M : \sigma$ and M is in *lnf*
by induction on the length of rewriting chain of $(\sigma, \Gamma) \rightsquigarrow M$.

Let $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha (n \geq 0)$,
$\Gamma' = \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n$.

$(\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma), n > 0$ must rewrite to $(\ \lambda\ x_1\ \ldots\ x_n\ .\ (\alpha, \Gamma')\ )$,
thus (even for $n = 0$) the chain must go through $(\ \lambda\ x_1\ \ldots\ x_n\ .\ (\alpha, \Gamma')\ )$.

(α, Γ') must rewrite to $(\ f\ (\rho_1, \Gamma')\ \ldots\ (\rho_m, \Gamma')\ )$
for some $(f, \rho_1 \to \ldots \to \rho_m \to \alpha) \in \Gamma'$,
thus the chain must go through $(\ \lambda\ x_1\ \ldots\ x_n\ .\ (\ f\ (\rho_1, \Gamma')\ \ldots\ (\rho_m, \Gamma')\ )\ )$.

After that each $(\rho_i, \Gamma') \rightsquigarrow M_i$ with shorter chain than $(\sigma, \Gamma) \rightsquigarrow M$.
Thus we have from the induction hypothesis for each M_i that
$\Gamma' \vdash M_i : \rho_i$ and M_i is in *lnf*.

$M = (\ \lambda\ x_1\ \ldots\ x_n\ .\ (\ f\ M_1\ \ldots\ M_m\ )\ )$ and each M_i is in *lnf*,
thus M is in *lnf*.

Following derivation shows how to derive $\Gamma \vdash M : \sigma$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\boxed{(f, \rho_1 \to \ldots \to \rho_m \to \alpha) \in \Gamma'}}{\Gamma' \vdash f : \rho_1 \to \ldots \to \rho_m \to \alpha} \quad \boxed{\Gamma' \vdash M_1 : \rho_1}}{\Gamma' \vdash (\ f\ M_1\ ) : \rho_2 \to \ldots \to \rho_m \to \alpha} \quad \boxed{\Gamma' \vdash M_2 : \rho_2}}{\vdots} \quad \ddots \quad \boxed{\Gamma' \vdash M_m : \rho_m}}{\Gamma' \vdash (\ f\ M_1\ \ldots\ M_{m-1}\ ) : \rho_m \to \alpha}}{\Gamma' \vdash (\ f\ M_1\ \ldots\ M_m\ ) : \alpha}}{\Gamma, x_1 : \tau_1, \ldots, x_{n-1} : \tau_{n-1} \vdash \lambda\, x_n\, .\, (\ f\ M_1\ \ldots\ M_m\ ) : \tau_n \to \alpha}}{\vdots}}{\Gamma, x_1 : \tau_1 \vdash \lambda\, x_2\ \ldots\ x_n\, .\, (\ f\ M_1\ \ldots\ M_m\ ) : \tau_2 \to \ldots \to \tau_n \to \alpha}}{\boxed{\Gamma \vdash \lambda\, x_1\ x_2\ \ldots\ x_n\, .\, (\ f\ M_1\ \ldots\ M_m\ ) : \tau_1 \to \tau_2 \to \ldots \to \tau_n \to \alpha}}$$

Now let us prove the opposite direction
$\Gamma \vdash M : \sigma$ and M is in *lnf* $\Rightarrow (\sigma, \Gamma) \rightsquigarrow M$
by induction on the size of M.

Let $M = \lambda\, x_1 \ldots x_n\, .\, f\ M_1\ \ldots\ M_m$,
$\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha$,
$\Gamma' = \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n$.

Then
$(\sigma, \Gamma) = (\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma) \longmapsto (\ \lambda\ x_1\ \ldots\ x_n\ .\ (\alpha, \Gamma')\ )$

by using the updated second grammar rule.

From $\Gamma \vdash M : \sigma$ we have that $(f, \rho_1 \to \ldots \to \rho_m \to \alpha) \in \Gamma'$ and $\Gamma' \vdash M_i : \rho_i$.
(This can be easily observed from definition of $\Gamma \vdash M : \sigma$.)

Therefore
$(\lambda\ x_1\ \ldots\ x_n\ .\ (\alpha, \Gamma')\ ) \longmapsto (\ \lambda\ x_1\ \ldots\ x_n\ .\ (\ f\ (\rho_1, \Gamma')\ \ldots\ (\rho_m, \Gamma')\ )\ )$
by using the first rule.

Each M_i is in *lnf*, is smaller in size then M and $\Gamma' \vdash M_i : \rho_i$;
therefore we have from the induction hypothesis that $(\rho_i, \Gamma') \rightsquigarrow M_i$.

Therefore $(\sigma, \Gamma) \rightsquigarrow (\ \lambda\ x_1\ \ldots\ x_n\ .\ (\ f\ M_1\ \ldots\ M_m\ )\ )$.

□

This means that we can use this grammar to generate all λ-terms M in *lnf* with desired type σ from *building blocks* context Γ.

2.5.1 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 [11], this method is obtained simply by replacing the first grammar rule by slightly modified rule, resulting in the following grammar.

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

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* [11]. 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.

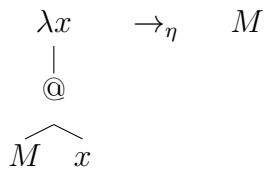
1. Every application of η-reduction shortens the term. Since every term has finite size, this process must end at some point. Thus every λ-term has *η-nf*.

2. Since η-reduction is *Church–Rosser*, *η-nf* is unique (see [9]).

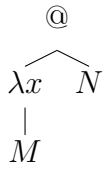
So we can take every generated λ-term M in *lnf* and transform it to shorter term in *η-nf*. The question is whether it remains in *β-nf*, thus being in *$\beta\eta$-nf*. The answer is yes; it can be proven by showing that no new β-redex is created by η-reduction.

Proposition 4. *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.

□

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 5. *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 2.4). Term M in $\beta\eta$-*nf* is in β-*nf*, thus it has desired *lnf* N which reduces to it.
Therefore it is surjection. □

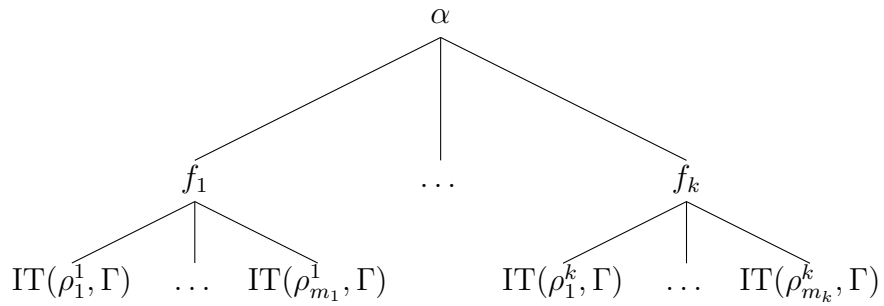
Suppose we have systematic (i.e. gradually generating all terms, but no term twice) method for generating terms in *lnf*, we may transform it to systematic method for generating terms in $\beta\eta$-*nf* by simply reducing each generated term to its η-*nf*.

2.6 Inhabitation tree

Now we introduce *inhabitation tree*; structure slightly different from *inhabitation machine*, which was introduced by Barendregt[11]. We can think about inhabitation tree as about unfolded inhabitation machine. Inhabitation tree approach is further refinement of the term generating grammar approach, which is too much textual. Whereas inhabitation tree approach aims at tree structure of λ-terms.

Definition. Let context Γ and type σ be given. *Inhabitation tree* for (σ, Γ), denoted as $\mathrm{IT}(\sigma, \Gamma)$, is a rooted possibly infinite tree that is recursively defined as follows.

If $\sigma = \alpha$ such that $\alpha \in A$, then
$\mathrm{IT}(\sigma, \Gamma) :=$



where
$\{ f_1 : \rho_1^1 \to \ldots \to \rho_{m_1}^1 \to \alpha \ , \ \ldots \ , \ f_k : \rho_1^k \to \ldots \to \rho_{m_k}^k \to \alpha \}$
$= \{ f \ : \rho_1 \to \ldots \to \rho_m \ \to \alpha \mid (f, \rho_1 \to \ldots \to \rho_m \to \alpha) \in \Gamma \}.$

Otherwise, for $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha$ such that $n > 0, \alpha \in A$
$\mathrm{IT}(\sigma, \Gamma) :=$

$$\begin{array}{c} \tau_1 \to \ldots \to \tau_n \to \alpha \\ | \\ \lambda x_1 \ \ldots \ x_n \\ | \\ \mathrm{IT}(\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n) \end{array}$$

Let us discuss this definition in grater detail. An inhabitation tree has two kinds of nodes.

1. *Type node* (or *OR-node*).

   This kind of node contains type.

2. *Symbol node* (or *AND-node*).

   This kind of node contains constant name, variable name or λ-*head* — non-empty finite sequence of variable names.

Following observation clarifies the alternation of node kinds.

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

There are two cases of σ — *atomic type* and *function type*.

First case is **atomic type**, i.e., $\sigma = \alpha$ where $\alpha \in A$.
For every $(f, \rho_1 \to \cdots \to \rho_m \to \alpha) \in \Gamma$ there is a child *symbol node* containing constant or variable name f. This symbol node containing f has m child subtrees corresponding to $\mathrm{IT}(\rho_1, \Gamma), \ldots, \mathrm{IT}(\rho_m, \Gamma)$.
Therefore specially for $m = 0$ node containing f is a leaf of the inhabitation tree.

Compare this case with corresponding grammar rule.

$$(\alpha, \Gamma) \longmapsto \left(\mathrm{f} \ \_ (\rho_1, \Gamma) \ \_ \ \ldots \ \_ (\rho_m, \Gamma) \right)$$

Second case is **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 that is not yet included in context Γ as variable or constant name.
There is one and only one child *symbol node* of function type node 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 $\mathrm{IT}(\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)$.

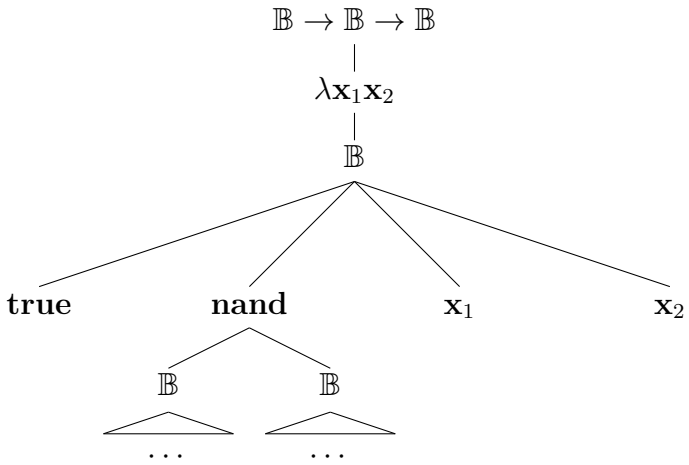
Compare this case with corresponding grammar rule.

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

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

$$\sigma = \mathbb{B} \to \mathbb{B} \to \mathbb{B}$$
$$\Gamma = \{\ true : \mathbb{B}$$
$$,\ nand : \mathbb{B} \to \mathbb{B} \to \mathbb{B}\ \}$$

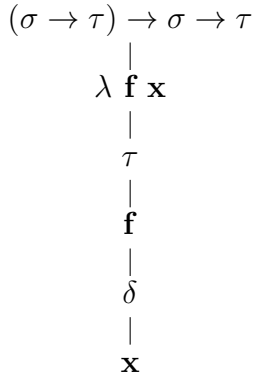
This particular (σ, Γ) results in the following tree:



Second example features our well known example:

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

Which results in following tree:



2.6.1 And-or tree and searching in Inhabitation tree

Let us consider following definition of *and-or tree*.

Definition.

1. *And-or tree* is a rooted possibly infinite tree with every node labeled as either *and-node* or *or-node*.
2. T' *solves* and-or tree T if T' satisfy following conditions.
   (a) T' is a finite tree, subgraph of T.
   (b) The root of T' is the root of T.
   (c) Each *and-node* in T' has all the child nodes as in T.
   (d) Each *or-node* in T' has precisely one child node.

Let us now consider following labeling of Inhabitation tree.

- **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. Or put in different words, it corresponds to filling all the slots for arguments with values.

2.6.2 Generating terms in *lnf* by solving inhabitation tree

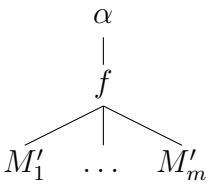
The motivation for defining *solving* and-or tree the way we did is that solution trees of $\mathrm{IT}(\sigma, \Gamma)$ are *typed sexpr-trees* representing all λ-terms M in *lnf* such that $\Gamma \vdash M : \sigma$.

Proposition 6. *(Correctness)*
If sexpr-tree M' solves $\mathrm{IT}(\sigma, \Gamma)$,
then M' represents λ-term M in lnf such that $\Gamma \vdash M : \sigma$.

Proof. We will prove following statement from which follows the rest by proposition 3: If sexpr-tree M' solves $\mathrm{IT}(\sigma, \Gamma)$, then M' represents λ-term M such that $(\sigma, \Gamma) \rightsquigarrow M$.

By induction on size of M'.

If $\sigma = \alpha$ such that $\alpha \in A$, then
$M' =$

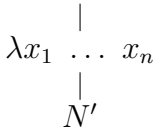


For some $f : \rho_1 \to \ldots \to \rho_m \to \alpha \in \Gamma$.

One can easily see that M'_i solves $\mathrm{IT}(\rho_i, \Gamma)$.
Thus by induction hypothesis we get that
M'_i represents λ-term M_i such that $(\rho_i, \Gamma) \rightsquigarrow M_i$.

$(\sigma, \Gamma) \longmapsto (\ f\ (\rho_1, \Gamma)\ \ldots\ (\rho_m, \Gamma)\ )$, thus $(\sigma, \Gamma) \rightsquigarrow (\ f\ M_i\ \ldots\ M_i\ )$.
Since M' represents $(\ f\ M_i\ \ldots\ M_i\ )$, the proof for the first case is complete.

Otherwise, for $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha$ such that $n > 0, \alpha \in A$
$M' = \tau_1 \to \ldots \to \tau_n \to \alpha$



Again, one can easily see that N' solves $\mathrm{IT}(\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)$.
Thus by induction hypothesis we get that
N' represents λ-term N such that $(\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n) \rightsquigarrow N$.

$(\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma) \longmapsto (\ \lambda\ x_1\ \ldots\ x_n\ .\ (\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)\ )$,
thus $(\sigma, \Gamma) \rightsquigarrow (\ \lambda\ x_1\ \ldots\ x_n\ .\ N\ )$.
Since M' represents $(\ \lambda\ x_1\ \ldots\ x_n\ .\ N\ )$, the proof is complete.

□

Proposition 7. *(Completeness)*
If M is λ-term in lnf such that $\Gamma \vdash M : \sigma$,
then there exists exactly one sexpr-tree M' such that M' represents M and M' solves $\mathrm{IT}(\sigma, \Gamma)$.

Proof. We will prove following statement from which follows the rest by proposition 3: If M is λ-term such that $(\sigma, \Gamma) \rightsquigarrow M$, then there exists exactly one sexpr-tree M' such that M' represents M and M' solves $\mathrm{IT}(\sigma, \Gamma)$.

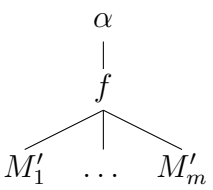
By induction on the length of rewriting chain of $(\sigma, \Gamma) \rightsquigarrow M$.

If $\sigma = \alpha$ such that $\alpha \in A$, then:
(σ, Γ) must rewrite to $(\ f\ (\rho_1, \Gamma)\ \ldots\ (\rho_m, \Gamma)\ )$.
And there is only one possibility for f since M is given.
$M = (\ f\ M_1\ \ldots\ M_m\ )$ where $(\rho_i, \Gamma) \rightsquigarrow M_i$.

From induction hypothesis we have that there exists exactly one sexpr-tree M_i' such that M_i' represents M_i and M_i' solves $\mathrm{IT}(\rho_i, \Gamma)$.

In order to construct M' solving $\mathrm{IT}(\sigma, \Gamma)$ we have only one possibility since we know the right f to take:

$M' =$



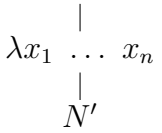
Otherwise, for $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha$, $(n > 0)$:

(σ, Γ) must rewrite to $(\ \lambda\ x_1\ \ldots\ x_n\ .\ (\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)\ )$.
$M = (\ \lambda\ x_1\ \ldots\ x_n\ .\ N\ )$ where $(\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n) \rightsquigarrow N$.

From induction hypothesis we have that there exists exactly one sexpr-tree N' such that N' represents N and N' solves $\mathrm{IT}(\alpha, \Gamma')$.

In order to construct M' solving $\mathrm{IT}(\sigma, \Gamma)$ in non-atomic σ we have only one possibility:

$M' = \tau_1 \to \ldots \to \tau_n \to \alpha$



□

2.6.3 In defense of inhabitation trees

It may seem that we are overthinking the problem by introduction of this new structure, whereas it would be sufficient to use the term generating grammars instead.

I see two reasons why inhabitation tree is useful concept.

1. The first reason is rather subjective. In my opinion, it simplifies the problem to be better graspable. My arguments are following:
   (a) It reduces the unpleasant fact that term generating grammars are not grammars in the standard sense (but instead *2-level-grammars*).
   (b) It brings the problem to its more natural domain, from (textual) grammars to (syntactical) trees.
   (c) We may think about the whole structure of generating at once since the whole tree, although usually infinite, is fairly simple structure of alternating layers of *and* and *or* nodes.
   (d) It enables us to ask such questions as: *What would be usage of inhabitation trees with number labeled edges coming from or-nodes?.*
2. We can think about the inhabitation tree as about some kind of search space. But another view is also possible: We can think about it as about collection of typed lambda terms that are generable by that inhabitation tree.

   This is even more visible when we introduce concept of an *incomplete inhabitation tree*; such a tree is created by omission of some subtrees with and-node roots (i.e., those accessible by *or* edge). *Finite incomplete inhabitation tree* corresponds to finite collection of typed λ-terms. In very natural sense we can call it a *typed multiterm* — a collection of typed terms forming one tree where those terms share the common parts.

   Thus the process of term generation is bidirectional — from (incomplete) inhabitation tree (seen as collection) we may obtain its element by solving it. But we may also take collection of typed trees and compress them into one tree by sharing their common parts, which happens to be the (incomplete) inhabitation tree that can be solved precisely by those terms which constructed it.

In the next chapter we will use A*-algorithm together with inhabitation tree ideas to perform systematic generation and other kinds of generation of λ-terms.

3. Design of our system

In this chapter we will describe our system trying to dive into implementation details as little as possible. Some of those details will be revealed in the next chapter devoted to the implementation.

This chapter will be presented in the mix between the mathematical terminology described in the previous chapter and the algorithmic terminology.

We can summarize our approach to design of the system for Genetic programming with types by its main design goals:

- It should eliminate the *closure* constrain of standard GP (see 1.2).
- It should be generalization of the standard GP described in 1, by which is meant that for *building blocks* satisfying constrains of standard GP there should be simple setup of control parameters of the system which will make the system behave in precisely the same way as the standard GP behaves.
- It should utilize the theory around typed λ-calculus.

3.1 Individuals and Building blocks

We have two competing variants for representing individuals — *@-tree* and *sexpr-tree* (see 2.1.6).

Individuals generated by solving inhabitation tree are sexpr-trees, which can be easily translated to @-trees.

Before we start generating we do one important trick. Suppose we want individuals of the type $\sigma = \tau_1 \to \ldots \to \tau_n \to \alpha$ from building blocks context Γ.

All our individuals will have form $\lambda\, x_1 \ldots x_n \,.\, M$. But since those variables are same for all individuals, we may do following trick.

Instead of generating individuals for $(\tau_1 \to \ldots \to \tau_n \to \alpha, \Gamma)$ we generate individuals for $(\alpha; \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n)$ and remember to add the λ-head $\lambda x_1 \ldots x_n$ to each individual when evaluating it.

This trick has two reasons:

1. Later in this chapter we will discuss problems occurring with variables when performing subtree swap in order to perform crossover. We solve those problems by performing *abstraction elimination*, which eliminates all occurences of variables and *lambda*-abstractions in a λ-term. But those problems are caused by local variables. Since $x_1 \ldots x_n$ are common for all λ-terms, it is valid to exclude them from this elimination. Since abstraction elimination prolongs size of λ-terms it is beneficial to lessen the effect of abstraction elimination in this way.

2. It also makes our method compatible with standard GP, where variables are treated as ordinary element of the set T and no λ-abstraction is used.

This ensures that S-expressions will be generated for Γ satisfying constrains of standard GP and that abstraction elimination will have no effect on such individuals.

3.2 Generating individuals

Input for term generating algorithm is pair (σ, Γ) where:

- σ is desired type of each generated individual.
- Γ is *building blocks* context.

Each generated λ-terms M must satisfy $\Gamma \vdash M : \sigma$.

We generate terms in long normal form. This is achieved by *solving* inhabitation tree for (σ, Γ). The main design choice behind our approach to solving inhabitation trees is ability to be variable enough to enable choice between *systematic* and *ramped half-and-half* (described in 1.3) methods of generating terms by choice of simple search strategy.

Let us explain what is meant by *systematic* method of generating terms.

Systematic way of solving inhabitation tree corresponds to generating terms in their long normal forms in order from smallest to largest in number of symbols and λ-heads.

We will achieve this by using *A* algorithm*.

3.2.1 A* algorithm

A* is a general informed search algorithm used for finding least-cost path from a given start state to some goal state in a state space. According to [13] A* is the most widely known form of best-first search.

By state space is meant oriented weighted graph with states as vertices and edges labeled with numbers corresponding to distance or cost. Edge from s_i to s_j means that state s_j is reachable from state s_i in one step with cost $dist(s_i, s_j)$. In pseudocode we will refer to set of states reachable from state s in one step as to $s.nexts()$.

Typically we use A* algorithm to find path from one specific start state to another specific goal state. But sometimes we can be interested only in finding a goal state which is specified by some *isGoal* predicate give to the algorithm as input (and the state space may contain many such goal states). This later variant corresponds to our situation, so we will continue by describing this variant.

As A* traverses the state space, it follows a path of the lowest expected total cost. The expectation is based on *heuristic* function which for state s predicts distance to the nearest goal state. It uses a priority queue of states as its crucial data structure. In this queue the priority of state s is the total expected cost of the path from start state going through s and continuing to the nearest goal state.

Here follows A* algorithm written in pseudocode:

```
function A*( start , isGoal , heuristic )
    open ← empty priority queue
    closed ← empty set

    start.G = 0
    start.F = 0 + heuristic( start )

    open.insert( start )

    while ¬ open.isEmpty() do
        state ← open.popStateWithLowestF()

        if isGoal( state ) then
            return state

        closed.insert( state )

        for next ∈ state.nexts() do
            newG ← state.G + dist(state,next)

            if (next ∉ open ∧ next ∉ closed) ∨ newG < next.G then

                next.G ← newG
                next.F ← newG + heuristic( next )

                if next ∉ open then
                    open.insert( next )

    return no-reachable-goal
```

Now we will modify this general A* algorithm so it will be more suited for our needs. We want following properties:

- The state space is a tree.
- $dist(s_i, s_j) = 1$
- We do not want to find one, but n goal states.

For state space which is a tree the condition ($next \notin open \wedge next \notin closed$) is always true, since it is possible to come to a state only from its parent state. This fact results in simplification of algorithm's code and behavior; we may completely omit the *closed* set and the checks in the for loop.

Those changes result in the following code:

```
function Our-A*( start , n , isGoal , heuristic )
    open ← empty priority queue
    results ← empty set

    start.G = 0
    start.F = heuristic( start )

    open.insert( start )

    while ¬ open.isEmpty() do
        state ← open.popStateWithLowestF()

        if isGoal( state ) then
            results.insert( state )
            if |results| = n then
                return results

        for next ∈ state.nexts() do
            next.G ←state.G + 1
            next.F ←next.G + heuristic( next )
            open.insert( next )

    return results
```

The most important part of A* algorithm is the heuristic function. A heuristic function must be *admissible* in order to make A* algorithm find the shortest possible path.

A heuristic function h is said to be *admissible* if it satisfies following condition for every state s:

$$h(s) \leq h^*(s)$$

where $h^*(s)$ is the true length of the shortest path from s to the nearest goal state.

In other words, a heuristic function is *admissible* if it is *optimistic*.

3.2.2 Our state space

In order to describe our state space to search in let us introduce extended version of our tree representation for typed λ-terms from 2.2.4. This extension consists in adding a new kind of leaf node (σ, Γ) standing for unfinished tree of a type σ with context Γ to use as set of building blocks.

Initial start states are all of a form "one node tree (σ, Γ)" since we are trying to generate a typed λ-term of a type σ from building blocks Γ.

A tree with no unfinished leaf is considered as final state.

We will define our state space inductively by specifying algorithm for obtaining successors of a state.

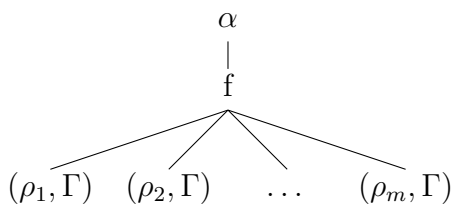
A non-final state must have one or more unfinished leafs. One of those unfinished leafs is selected. It is the first one found by depth-first search (from the root of a state tree) i.e., the leftmost unfinished leaf (if we consider that tree written as expression).

Let (σ, Γ) be the selected leaf. Successor state is constructed by replacing this leaf by new subtree. We must distinguish two cases of σ : *Atomic type* σ and *function type* σ.

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

For atomic σ there are as many successors as there are members of Γ of a form $(f, \rho_1 \to \cdots \to \rho_m \to \boldsymbol{\alpha})$, where $m \geq 0$. In other words all the members of Γ of a type α or of a type for function which returns α.

For each $(f, \rho_1 \to \cdots \to \rho_m \to \boldsymbol{\alpha}) \in \Gamma$, where $m > 0$ the new subtree which will replace (σ, Γ) has following form:



And for m = 0:

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

For function type σ there is exactly one successor.

For every $i \in \{1, \ldots, n\}$ we create new *variable name* x_i which is not yet used anywhere in the whole tree.

The new subtree which will replace (σ, Γ) has following form:

$$\begin{array}{c} \tau_1 \to \cdots \to \tau_n \to \alpha \\ | \\ \lambda x_1 \ldots x_n \\ | \\ (\alpha, \Gamma \cup \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}) \end{array}$$

We should discuss that it is correct to have as successors of a state those created by expansion of the leftmost unfinished leaf (in contrast with more broad successor set where any unfinished leaf may be expanded). What we need to show is that no final state will be omitted by the method and that there is no shorter path omitted.

One can see that it is correct by considering that the order of expanding

unfinished leafs is irrelevant since expansion of each unfinished leaf is independent from expansion of any other unfinished leaf (up to renaming of variables which is irrelevant).

Every unfinished leaf of a state must be replaced by tree containing no unfinished leafs in order to be final state. But since expansion of an unfinished leaf has no impact (besides new variable names) on expansion of other unfinished leafs we may choose the order of expansions arbitrarily.

We prefer this variant because:

- It has smaller or equal number successors for each state, i.e. smaller or equal branching factor.
- It makes the state space a tree, in contrast with the naive state space which is not a tree.

3.2.3 Our heuristic function

A heuristic function takes as input a state and returns as output an estimation of number of steps needed to get to a final state by the shortest possible path.

Our heuristic function is a simple one: The number of unfinished leafs in the state tree.

This heuristic function is admissible because every unfinished leaf must be at least once expanded to become final.

3.2.4 Further improvements

We may easily put all deterministic expansion steps into one step — by which we mean the lambda head expansion. This could be done when expanding atomic type by also expanding its non-atomic arguments. This would result in that only atomic types wold by in unfinished leafs. It is important to accordingly change the *dist* in A* which is now 1 generally.

It is also worth mentioning that *dist* 1 is also for lambda heads with long list of variable names in the current specification — we do not bother ourselves with this detail. But when it is not the desirable behavior it is very easy to change by using mor complicated *dist* than 1.

3.2.5 Search strategy

Now we will describe further modification of the A* algorithm which is generalizing it enough to enable use of *ramped half-and-half* term generating method and other term generating methods.

It is done by adding one more input argument for the algorithm. This additional argument is a search strategy whose purpose is to determine which successor states of all possible successor states will be added to the priority queue, i.e., a search strategy is there to filter out some successor states from $state.nexts()$. This filtering may be non-deterministic. It is based on the depth of expanded unfinished leaf and on properties of the newly added subtree, e.g., whether the

newly added subtree contains one or more unfinished leafs (so it corresponds to non-terminal) or not (so it corresponds to terminal).

By *depth* we really mean $\lfloor depth/2 \rfloor$, because our formal state trees contain those type nodes above each symbol node.

In previous situation where there is no filtering of states involved we know that if queue becomes empty, then we have exhaustively searched all possible states.

But since filtering is involved this is no longer true. Therefore, if the queue becomes empty before the required number of terms is generated, then the generating process is restarted and it continues to generate terms.

We also enable strategy to have some internal state which is initialized (possibly non-deterministically) each time the generating process restarts.

Hopefully those changes will be clarified by the following pseudocode:

```
function A*-with-strategy( start , n , isGoal , heuristic , strategy )
    results ← empty set

    while |results| < n do
        strategy.init()

        open ← empty priority queue

        start.G = 0
        start.F = heuristic( start )

        open.insert( start )

        while ¬ open.isEmpty() do
            state ← open.popStateWithLowestF()

            if isGoal( state ) then
                results.insert( state )
                if |results| = n then
                    return results

            nexts ← strategy.filter( state.nexts() , state.depth() )

            for next ∈ nexts do
                next.G ←state.G + 1
                next.F ←next.G + heuristic( next )
                open.insert( next )

    return results
```

Let us show examples of search strategies. They are described by describing their $filter(nexts, depth)$ and $init()$ methods[1] . Again, in order to avoid confusion, let us stress that *depth* is really $\lfloor depth/2 \rfloor$ since our formal states contain above each symbol node a type node.

Systematic strategy

The previous systematic behavior may be obtained by trivial $filter$ function which returns all *nexts*. Systematic strategy needs no internal state, therefore *init* is doing nothing.

Ramped half-and-half strategy

In the *init* are randomly (uniformly) initialized two variables of the strategy internal state:

- *isFull* - A boolean value, determining whether *full* or *grow* method will be performed.
- d - A integer value from $\{2, \ldots, D_{init}\}$, where D_{init} is predefined maximal depth (e.g. 6).

The method $filter(nexts, depth)$ returns precisely one randomly (uniformly) selected element from a subset of *nexts* (or zero elements if *nexts* is empty). This means that the queue always contains only one (or zero) state.

The subset to select from is determined by $depth, d$ and $isFull$.

Those elements of *nexts* whose newly added subtree contains one ore more unfinished leafs are regarded as *non-terminals*, whereas those whose newly added subtree contains no unfinished leaf are regarded as *terminals*.

If $depth = 0$, then the subset to select from is set of all *non-terminals* of *nexts*.
If $depth = d$, then the subset to select from is set of all *terminals* of *nexts*.

In other cases of *depth* it depends on value of $isFull$.
If $isFull = true$, then the subset to select from is set of all *non-terminals* of *nexts*.
If $isFull = false$, then the subset to select from is whole *nexts* set.

Geometric strategy

We can see those two previous strategies as two extremes on the spectrum of possible strategies.

Systematic strategy filters no successor state thus performing exhaustive search resulting in discovery of n smallest terms in one run.

On the other hand, *ramped half-and-half strategy* filters all but one successor states resulting in degradation of priority queue to "fancy variable". And each

[1]It is also needed to add forceUniqueness() boolean predicate to the strategy since Koza's ramped half and half requires uniqueness. Let us skip this step here in this explanation,since it can be easily implemented but adds lot of unnecessary distraction. May the reader forgive.

term is generated in its own run.

Geometric strategy is simple yet fairly effective term generating strategy somewhere in the middle of this spectrum.

It is parameterized by parametr $q \in (0, 1)$, its default well-performing value is $q = 0.75$.

For each element of *nexts* it is probabilistically decided whether it will be returned or omitted. A probability p of returning is same for all elements, but depends on *depth*. It is computed as follows:

$$p = q^{depth}$$

This formula is motivated by idea that it is important to explore all possible root symbols, but as the *depth* increases it becomes less "dangerous" to omit an exploration branch.

We can see this by considering that this strategy results in somehow undisciplined A* search. With each omission we make the search space smaller. But with increasing depth these omissions have smaller impact on the search space, i.e., they cut out lesser portion of the search space.

Another slightly esoteric argument supporting this formula is that "root parts" of a program usually stand for more crucial parts with radical impact on global behavior of a program, whereas "leaf parts" of a program usually stand for less important local parts (e.g. constants).

It also plays nicely with the idea that "too big trees should be killed".

Systematic strategy does not have an internal state, thus *init* does nothing.

3.2.6 Final remarks

We should mention that since simply typed lambda calculus is equivalent to implicational fragment of intuitionistic logic it wold be possible to do term generation by prover for this logic. But logic prover is concerned with finding one proof of a difficult proposition, whereas types usually tend to be easy propositions but we need enormous amount of them. So this two fields, although equivalent in theory do not have equivalent goals in practice.

We have also tried generation directly from grammar rules corresponding to basic inference rules – we did it before encountering the work of [11], but since this has two drawbacks we have not supported the implementation of the former term generator to the actual state of source code.

Those drawbacks are that it is not in normal form and that it is not obvious which type shoud by added in the grammar rule corresponding to the $\rightarrow$*-elimination*.

3.3 Crossover

Design goal behind our approach to crossover operation is to try to generalize standard tree swapping crossover. This is not because of dislike of more specialized crossovers; it comes from systematic approach for exploring the domain of typed functional GP — thus I would like to start with the most common one.

The crossover operation in standard GP is performed by swapping randomly selected subtrees in each parent S-expression (see 1.6).

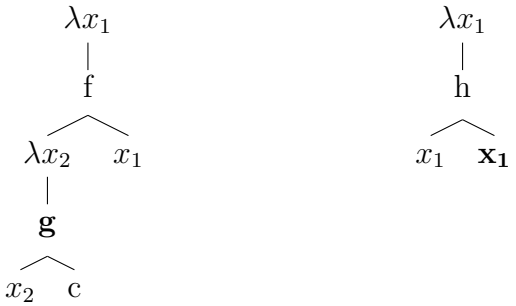
For typed lambda terms two difficulties arise: Types and variables.

We will show how to crossover *typed λ-term trees* in both @-tree and sexpr-tree notation.

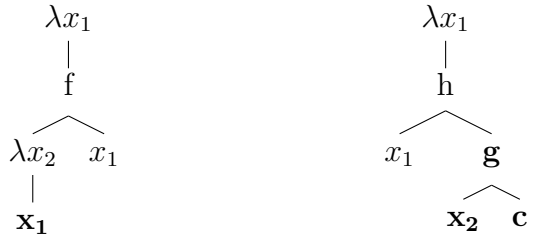
As in standard GP our crossover will be performed by swapping two subtrees. But now with constraint that both subtrees have the same type. This is discussed in grater detail in 3.3.2.

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.

First approach tried at this problem was to repair the broken term, but since this method is not currently implemented we will describe it in just a few words. If the moved free variable ends in the scope where is defined variable with the same name and type, then it is OK. If it is not this case, we may rename this variable to some other variable "visible" in its new scope if such a variable has the same type. If there is no such a suitable variable we may generate new value

to replace the variable by term generating method.

As you can see, this is not much elegant solution.

But we can avoid dealing with this problem by avoiding use of variables. This can be achieved by process called abstraction elimination.

3.3.1 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:

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

Let us describe transformation Æ performing this process.[2]

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

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 [12].

This version operates with more combinators:

$$\begin{aligned}\mathbf{B} &= \lambda\, f\, g\, x\,.\, f\, (g\, x)\\ \mathbf{C} &= \lambda\, f\, g\, x\,.\, f\, x\, g\\ \mathbf{S}' &= \lambda\, c\, f\, g\, x\,.\, c\, (f\, x)\, (g\, x)\\ \mathbf{B}* &= \lambda\, c\, f\, g\, x\,.\, c\, (f\, (g\, x))\\ \mathbf{C}' &= \lambda\, c\, f\, g\, x\,.\, c\, (f\, x)\, g\end{aligned}$$

[2] In our implementation we must also deal with types, because our individuals are annotated with types. But since this process is fairly straightforward but cumbersome to describe, we will skip explaining it here. Reader interested in seeing it can see the source code of our implementation.

And the transformation can be written as follows.

$$
\begin{aligned}
Æ[x] &= x \\
Æ[\,(M\ N)\,] &= (Æ[M]\quad Æ[N]) \\
Æ[\lambda\, x\,.\, M] &= A[x; Æ[M]]
\end{aligned}
$$

$$
\begin{aligned}
A[x; x] &= \mathbf{I} \\
A[x; y] &= (\mathbf{K}\ y) \\
A[x;\ (M\ N)\,] &= Opt[\ \mathbf{S}\ (A[x; M])\ (A[x; N])\ ]
\end{aligned}
$$

$$
\begin{aligned}
Opt[\ \mathbf{S}\ (\mathbf{K}\ M)\ (\mathbf{K}\ N)\ ] &= \mathbf{K}\ (M\ N) \\
Opt[\ \mathbf{S}\ (\mathbf{K}\ M)\ \mathbf{I}\ ] &= M \\
Opt[\ \mathbf{S}\ (\mathbf{K}\ M)\ (\mathbf{B}\ N\ L)\ ] &= \mathbf{B}*\ M\ N\ L \\
Opt[\ \mathbf{S}\ (\mathbf{K}\ M)\ N\ ] &= \mathbf{B}\ M\ N \\
Opt[\ \mathbf{S}\ (\mathbf{B}\ M\ N)\ (\mathbf{K}\ L)\ ] &= \mathbf{C}'\ M\ N\ L \\
Opt[\ \mathbf{S}\ M\ (\mathbf{K}\ N)\ ] &= \mathbf{C}\ M\ N \\
Opt[\ \mathbf{S}\ (\mathbf{B}\ M\ N)\ L\ ] &= \mathbf{S}'\ M\ N\ L
\end{aligned}
$$

As is stated in [12], 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.

3.3.2 Typed subtree swapping

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

4. Implementation

In this chapter we will briefly discus some parts of the implementation of our system. Some previous knowledge of Haskell is assumed, but one can freely skip this chapter since it is not crucial for the overall message of this thesis.

Our system is implemented in functional language *Haskell*. Program implementing it is called *Fishtron*. It runs as a simple server application and its GUI is accessible through web browser.

Evaluation of individuals is performed by translating them from λ-terms into Haskell programs which are evaluated in the Haskell interpreter *Hint* accessible in the code as `Interpreter` monad.

Each problem definition typically consists of two Haskell source code files. First file (usually called `Problem.hs`) contains problem definition involving such things as desired type of individual, context of building blocks, fitness function implementation and various control parameters like term generating method, crossover, default number of runs, generations, population size etc.

Those functions or constants of building block context that are not part of the standard Haskell *Prelude*[1] need to be defined in the second file (usually called `Funs.hs`).

After start of the program one can access the interface by pointing a browser to `http://localhost:3000`. Here is simple form where registered problems are listed. Beside problem selection one can also change some control parameters. After hitting the "start" button evolution begins. From that point until the end of the running experiment, the front-end JavaScript program periodically asks server for new run information. Server makes this information available at the end of each generation. This information contains various data: Haskell and JavaScript source codes of the best individual, some statistical data about the just evaluated generation, standard output of the server program, etc.

These data are immediately available in the form of continuously updated graphs and if the *phenotype* JavaScript file is present for the problem, then is also available immediate presentation of the best individual from each generation. JavaScript functions standing for best individuals are also immediately accessible through JavaScript console, e.g. for debugging purposes.

Now let us move our attention to the architecture of the evolution engine. Various dirty parts of the outside world (such as textual output, communication with front-end, access to random number generator) are packed into `Eva` state monad defined in `Eva.hs`.

Core of the evolution engine (defined in `GP_Core.hs`) is done so it enables various structures to be evolved - such as sexpr-trees, @-trees or even boolean lists of fixed length as to support beside GP also Genetic Algorithms.

[1] A standard module imported by default into all Haskell modules.

This is possible due to following type classes.

```
class Gene term opt where
 generateIt ::  Int -> opt -> Eva [term]

class Muta term opt where
 mutateIt ::  opt -> term -> Eva term

class Cros term opt where
 crossIt ::  opt -> term -> term -> Eva (term,term)

class Evolvable term a gOpt mOpt cOpt where
 evolveIt ::  RunInfo -> Problem term a gOpt mOpt cOpt
                      -> Eva (term,FitVal,Maybe Int)
```

Put simply, if we define instances for `Gene`, `Muta` and `Cros`, then we get instance of `Evolvable`.

`Gene` correspond to individual generating method for structure `term` parametrized by option `opt`. Analogically for mutation and crossover.

Instances of those three type classes are located in the file `GP_Data.hs`.

We will skip deeper explanation of this topic now.

One technical detail that I think is worth mentioning here is a trick involved in evaluation of Haskell programs corresponding to individuals. Straightforward evaluation technique would be to simply call evaluation function on every string corresponding to individual expression.

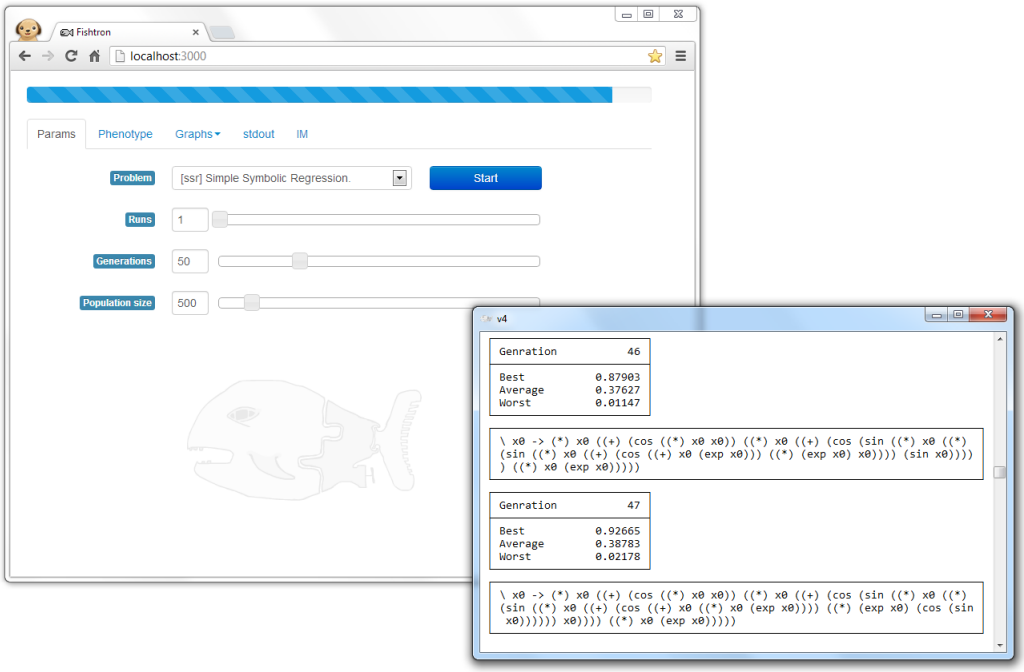
But since most time consuming activity of evaluating using *Hint* is its initialization, it is much more effective to collect all the individuals to be evaluated in the current generation and create huge string representing list of expressions and evaluate this string of all individual.

Another notable comment about implementation comes from fact that *Hint* needs to know type of the expression being evaluated. This particular thing had enormous enlarging effect on pain produced by need to fight the type system in order to make it reasonably working piece of software. This may be caused by the fact that I am still pretty inexperienced Haskell programmer, since this is my first bigger program done in this language.

Moral of this remark is the following one: If you plan to use the *Hint* in your Haskell program on place where it will be used to eval expressions of various types, think through the integration of the evaluator into the whole system in advance.

Let us finish this brief chapter about implementation of our system by listing other not yet mentioned source files with a little information about them.

- `Server.hs` — File containing main of the server.
- `Register.hs` — Place for registering the problem to make it appear in the list of problems in GUI.
- `InhabTree.hs` — Core of the term generating method.
- `TTerm.hs` — Code related to typed λ-terms represented as @-trees.
- `TTree.hs` — Code related to typed λ-terms represented as sexpr-trees.
- `Heval.hs` — Evaluator wrapper code.
- `Dist.hs` — Implementation of "probability distribution" data structure used as container for population.



5. Results

In this section will be presented usage of the system in order to solve specific problems. Several problems will be described and some of them compared with results of others.

5.1 Simple Symbolic Regression

Simple Symbolic Regression is a problem described in [1]. Objective of this problem is to find a function $f(x)$ that fits a sample of twenty given points. The target function is function $f_t(x) = x^4 + x^3 + x^2 + x$.

Desired type of generated programs σ and building blocks context Γ are following.

$$\begin{aligned}\sigma &= \mathbb{R} \to \mathbb{R} \\ \Gamma = \{&(+) : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, \\ &(-) : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, \\ &(*) : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, \\ &rdiv : \mathbb{R} \to \mathbb{R} \to \mathbb{R}, \\ &sin : \mathbb{R} \to \mathbb{R}, \\ &cos : \mathbb{R} \to \mathbb{R}, \\ &exp : \mathbb{R} \to \mathbb{R}, \\ &rlog : \mathbb{R} \to \mathbb{R}\}\end{aligned}$$

where

$$rdiv(p, q) = \begin{cases} 1 & \text{if } q = 0 \\ p/q & \text{otherwise} \end{cases}$$

$$rlog(x) = \begin{cases} 0 & \text{if } x = 0 \\ log(|x|) & \text{otherwise.} \end{cases}$$

Fitness function is computed as follows

$$fitness(f) = \sum_{i=1}^{20} |f(x_i) - y_i|$$

where (x_i, y_i) are 20 data samples from $[-1, 1]$, such that $y_i = f_t(x_i)$.

An individual f such that $|f(x_i) - y_i| < 0.01$ for all data samples is considered as a correct individual.

5.1.1 Experiments

On this problem we demonstrate that if control parameters are set to be compatible with standard GP, then the results obtained fit the results presented in [1] relatively well.

Those control parameters are following two:

- *Ramped half-and-half strategy* is used as search strategy.
- Option of preserving the best individual into the next population is set **off**.

The experiment consisted of 50 independent runs of GP algorithm. Each run had maximally 50 generations (50 + 1 for generation 0) and 500 individuals as population size.

We analyze the ability of the system to produce a correct solution. We are interested in percentage of runs that led to discovery of correct solution. And we also use the popular measurement method within GP field — the *performance curves* described in [1].

First performance curve is the cumulative *probability of success* $P(M, i)$, where i is generation number ($i = 0$ for initial population) and M is the size of the population. $P(M, i)$ is probability that a correct solution will be find in one run of GP with maximal number of generations i and population size M. $P(M, i)$ is computed as percentage of runs yielding a correct solution in generation i or earlier.

From values of $P(M, i)$ are computed values of the second performance curve $I(M, i, z)$ — the total number of individuals that must be processed to yield a correct solution with probability z by multiple runs of GP with maximal number of generations i and population size M (usually $z = 0.99$).

As is described in [1], The number $R(M, i, z)$ of independent runs required to yield a correct solution is computed as

$$R(M, i, z) = \lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \rceil,$$

thus $I(M, i, z)$ is computed as

$$I(M, i, z) = (i + 1) \cdot M \cdot R(M, i, z).$$

It scored 17/50 (34%) success rate. Minimal $I(M, i, z)$ was in generation 23 with 192,000 individuals to be processed. The experiment took 46 minutes.

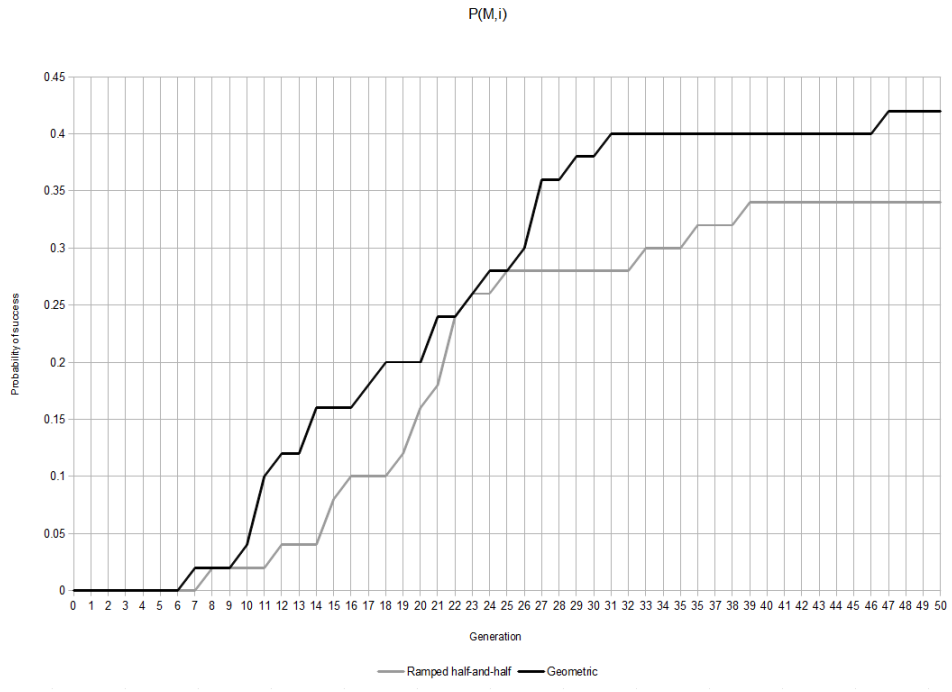
Let us compare it with results from [1] (based on 113 runs):

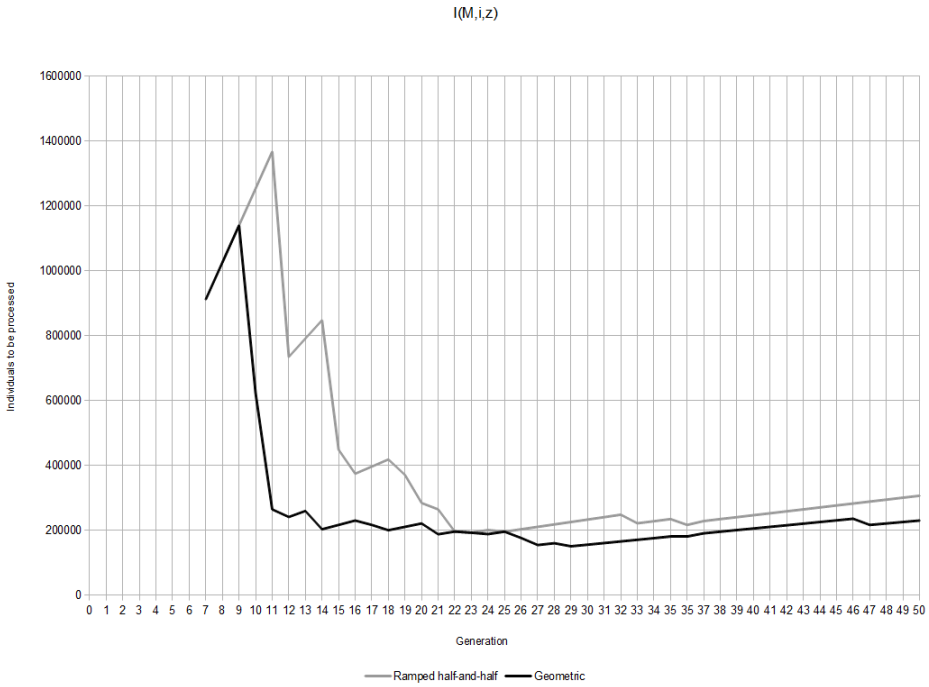
Success rate 35%; minimal $I(M, i, z)$ in generation 24 with 162,500 individuals to be processed. This seems like a pretty good match.

Second experiment was performed to compare *ramped half-and-half strategy* with our *geometric strategy* (with default parameter $q = 0.75$). Thus all control parameters stay same except for the search strategy for generating algorithm.

It scored 21/50 (42%) success rate. Minimal $I(M, i, z)$ was in generation 29 with 150,000 individuals to be processed. The experiment took 26 minutes.

So our geometric strategy is slightly more successful then ramped half-and-half strategy. But more interesting is that it made the whole experiment almost twice as fast; reason of this will be discussed in more detail in the next problem.

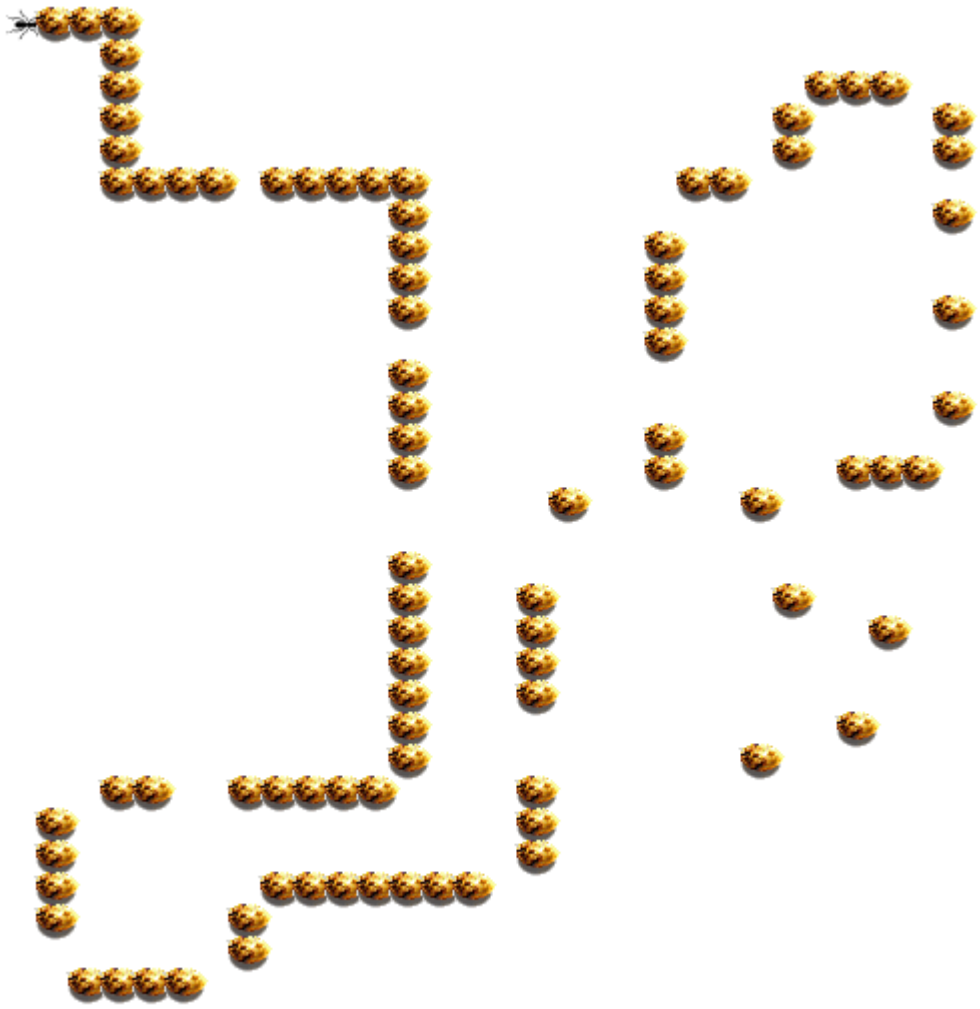




5.2 Artificial Ant

Artificial Ant is another problem described in [1]. Objective of this problem is to find a control program for an artificial ant so that it can find all food located on "Santa Fe" trail.

The Santa Fe trail lies on toroidal square grid. The ant is in the upper left corner, facing right. It has following layout.



The ant is able to move forward, turn left, and sense if a food piece is ahead of him.

Desired type of generated programs σ and building blocks context Γ are following.

$$\begin{aligned}\sigma &= AntAction\\ \Gamma = \{\ \ l &: AntAction,\\ r &: AntAction,\\ m &: AntAction,\\ ifa &: AntAction \to AntAction \to AntAction,\\ p2 &: AntAction \to AntAction \to AntAction,\\ p3 &: AntAction \to AntAction \to AntAction \to AntAction\}\end{aligned}$$

Action l turns the ant left.

Action r turns the ant right.

Action m moves the ant forward.

Action $ifaxy$ (if-food-ahead) performs action x if a food piece is ahead of the ant, otherwise it performs action y.

Action $p2xy$ first performs action x and after it action y.

Action $p3xyz$ first performs action x, after that action y and finally z.

Actions l, r and m each take one time step to execute.

Ants action is performed over and over again until it reaches predefined maximal number of steps. In [1] this limit is set to be 400 time steps.

Fitness value is equal to number of eaten food pieces.

An individual such that eats all 89 pieces of food is considered as a correct solution.

There is also mentioned following ant program which is described as correct solution.

```
(ifa m (p3 l (p2 (ifa m r) (p2 r (p2 l r)))(p2 (ifa m l) m)))
```

But here arises a problem: In our implementation this program needs 545 time steps; if it is given only 400 time steps, then it eats only 79 pieces of food. It is uncertain whether it is caused by my mistake in implementation or by inaccuracy in [1]. Since everything else seems working just fine, I have decided to set the limit to 600 in my experiments.

5.2.1 Experiments

Due to those uncertainties, we will not perform comparing experiments as we did in the problem of simple symbolic regression.

In the following experiments, the option of preserving the best individual into the next generation is active (since we are not trying to compare with results in [1]).

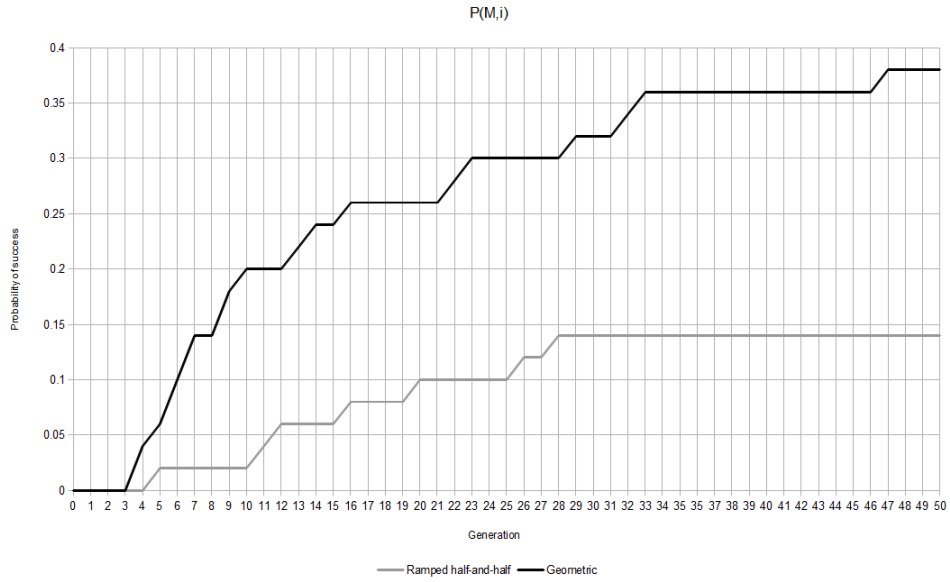
As usually, first experiment consisted of 50 independent runs of GP algorithm. Each run had maximally 50 generations and 500 individuals as population size.

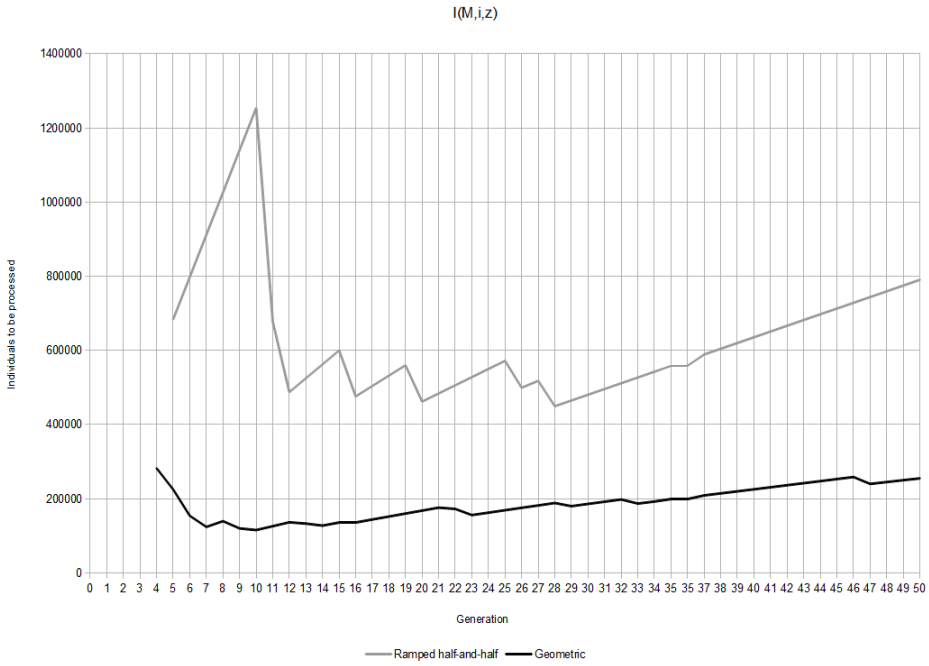
Ramped half-and-half strategy is used as search strategy for the first experiment.

It scored 7/50 (14%) success rate. Minimal $I(M, i, z)$ was in generation 28 with 449,500 individuals to be processed. The experiment took 265 minutes.

Similarly as in previous problem, second experiment was performed to compare *ramped half-and-half strategy* with our *geometric strategy* (with default parameter $q = 0.75$). Thus all control parameters stay same except for the search strategy for generating algorithm.

It scored 19/50 (38%) success rate. Minimal $I(M, i, z)$ was in generation 10 with 115,500 individuals to be processed. The experiment took 107 minutes.





This is big improvement in all three watched factors:

1. 38% vs 14% success rate.
2. 115,500 vs 449,500 individuals needed to be processed.
3. 107 vs 265 minutes.

Third factor, the time difference might give us clue about what is going on.

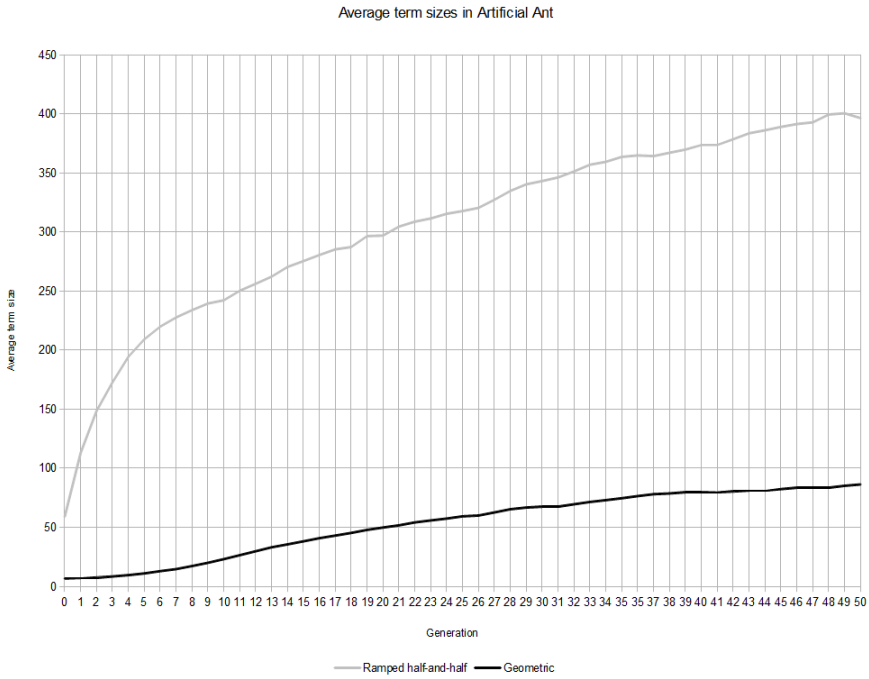
Those two experiments differs only in used term generating method. During one run the vast majority of the time is spend by evolution, the generation process only slightly prolongs the time spend in generation 0. The run time of the evolution is influenced by two important factors: Number of generations and time that crossover takes.

The number of generations of more successful method is surly smaller, but it seems improbable that it would have such a big impact (this is idea is supported by results of experiments with even-parity problem, where times are almost the same but success rates differs significantly).

Time of crossover is proportionate to number of nodes that an individual has. Thus it seems that observed slowdown is due to terms being too big.

In order to test this hypothesis, we run additional experiments to get statistics about dynamics of individuals' sizes.

Following graphs are base upon 10 run experiments.



Those graphs show enormous differences. The graph of Ramped half-and-half strategy evidences occurence of well known problem of GP called *bloat*. Bloat is defined as *program growth without (significant) return in terms of fitness* [3].

As is further stated in [3], bloat has significant practical effects.

- Large programs are computationally expensive to evolve and run.

- Large programs can be hard to interpret.
- Large programs may exhibit poor generalisation.

For these reasons bloat has been a subject of intense study in GP.

On the contrary, our strategy seems to be successfully fighting with bloat in this particular problem.

It is also noticeable from the correct solutions that have been found.

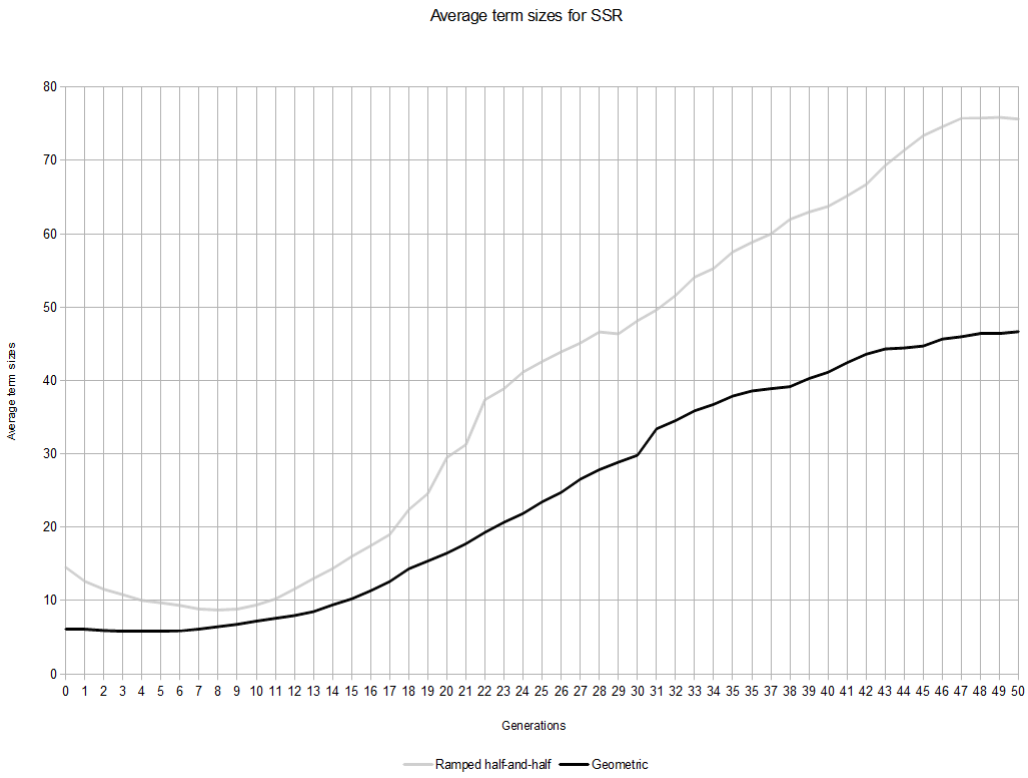
Here follow all 7 correct solutions found using ramped half-and-half strategy.

1. `p2 (ifa (p2 l r) (ifa (ifa (p2 m m) (p2 m l)) (p2 (ifa l r) (p2 m r)))) (ifa (i fa (p3 m (ifa m m) l) (p2 r r)) (ifa (ifa (p3 (p2 m l) (ifa (p3 (p2 m (ifa l m)) (p3 (ifa r (p2 (ifa (p2 m r) (p3 l r m)) (p2 m r))) m l) (p2 m l)) (ifa (p3 r l l) (p3 m m r))) (ifa (p3 l l r) (p2 (ifa l r) (p2 m r)))) (p2 l r)) (ifa (p3 m l l) (p2 r r))))`
2. `p3 (ifa (p3 (p3 m m r) (p3 (ifa (p3 (p3 m m r) (p3 r l l) (ifa m l)) (ifa (p3 l r (ifa (ifa m m) (p3 r l (p2 (ifa m l) (ifa (p3 (p2 l (p3 (ifa m m) (p2 l m) (p2 l r))) (p2 (ifa (ifa m r) (p3 r m l)) (p3 (ifa l l) (p3 l r l) (p2 l l))) (p2 (ifa (ifa m l) (p2 l r)) (p2 (p2 l m) (p3 l r m)))) (ifa (ifa (p3 (p3 r m m) (p3 (p2 l m) (p2 l m) (ifa m m)) (ifa l m)) (ifa (p3 l r m) (ifa (ifa r r) (p2 m l)))) l)))))) (ifa l l))) m (p2 (p2 (ifa l l) (ifa m r)) r)) (ifa m l)) (ifa (p3 l r (ifa (ifa m m) (p3 r l (p2 (ifa m l) (ifa (p3 (p2 l (p3 (ifa m m) (p2 l m) (p2 l r))) (p2 (ifa (ifa m r) (p3 r m l)) (p3 (ifa l l) (p3 l r l) (p2 l l))) (p2 (ifa r (p2 l r)) (p2 (p2 l m) (p3 l r m)))) (ifa (ifa (p3 (p3 r m m) (p3 (p2 l m) (p2 l m) (p2 l r)) (ifa l m)) (ifa (p3 l r m) (ifa (ifa r r) (p2 m l)))) l)))))) (ifa l l))) m (p2 (p2 (ifa l l) (ifa m r)) r)`
3. `p2 (ifa (p3 m m m) (p2 r m)) (p3 (ifa l r) (ifa (ifa l (p3 (p3 r r l) (ifa l m) (ifa m (ifa (p2 (p3 (p3 (p3 (ifa m l) (p2 r r) (ifa m m)) (p3 (p3 r r r) (ifa r r) (p2 m l)) (p3 (p2 m m) (p3 r m r) (ifa m r))) (p2 (p2 (p3 r l m) (p2 m m)) (p3 (ifa r l) (p2 m l) (p2 r m))) (p3 (ifa (ifa r l) (p3 m m r)) (p3 (ifa r m) (p3 l l r) (p2 r r)) l)) (p2 (p2 (p2 (p2 r r) (p2 r l)) (p3 (ifa r r) (p2 l l) (p2 m l))) (p3 m r l))) (p2 (ifa (ifa (p3 (ifa m r) (p2 (p3 (p3 r r l) (ifa l m) (p3 m l r)) (p2 m r)) (p3 l l l)) (ifa (p3 l l m) (ifa l r))) (ifa (ifa (ifa m r) (ifa l l)) (p2 (ifa l m) (p2 l m)))) (p3 (ifa (p2 (p2 r m) (p3 m l m)) (ifa (p2 r l) (p3 r m r))) (p2 (p2 (p2 r r) (ifa l r)) (p3 (p3 m m m) (ifa m r) (ifa l m))) (p2 m l))))))) r) (ifa l r))`
4. `p2 (p3 (p2 m l) (ifa r l) (ifa m l)) (ifa (p2 m m) (ifa r l))`
5. `p2 (p2 m r) (ifa m (p2 (p2 r r) (ifa (ifa (ifa m r) (p3 m m l)) r)))`
6. `ifa (p2 (p2 m m) (p2 m l)) (p3 (p2 r r) (ifa m l) (p2 m l))`
7. `p2 (ifa (p2 (ifa r m) (p2 (p2 (p2 l m) (p3 m m r)) (ifa l l))) (p3 (p2 l l) (ifa m l) m)) l`

Compare it with 18 solutions obtained by using geometric strategy (I accidentally lost one solution).

1. `ifa m (p3 r (ifa (ifa m (ifa m (p3 r (ifa m r) m))) (ifa m (p2 (p2 m (ifa m (ifa m (ifa m (p3 r (ifa m r) m))))) (ifa m r)))) m)`
2. `p3 (ifa m l) (p3 m l (ifa (p2 m (ifa l (p2 (p2 (p2 m (ifa m m)) (ifa (ifa m l) r)) (ifa m l)))) r)) r`
3. `ifa l (ifa (p3 (p2 (p2 r (ifa r m)) (p3 l (ifa m (ifa m l)) r)) l m) (p2 r (ifa m (p2 l (p2 (p2 l (ifa m r)) m)))))`
4. `p3 m (p3 r (ifa m l) l) (ifa (p2 (p3 m m r) (ifa m l)) r)`
5. `p3 (ifa m (ifa (ifa m m) (ifa m r))) m (p3 r (ifa (p3 (ifa m r) m (p3 r (ifa m l) m)) l) l)`
6. `p3 (ifa m l) (p2 m l) (ifa (p3 (ifa m (ifa m l)) m r) (p2 (ifa m (ifa r l)) l))`
7. `p3 (p3 r (ifa (p2 (ifa m (ifa (ifa m r) m)) m) l) m) l (ifa m r)`
8. `ifa (p3 m (ifa (ifa r m) r) l) (p3 (p3 l (ifa m r) r) (ifa m (ifa m l)) m)`
9. `p3 m (p3 l (ifa (p3 m (ifa m m) m) r) r) (ifa m (ifa m l))`
10. `p2 (ifa (ifa m (p3 l (p2 (p2 (ifa (ifa m l) l) m) (ifa m r)) (p2 (ifa (ifa (ifa (ifa (p3 r l m) r) l) l) (p3 l (ifa m r) (ifa m r))) m))) r) (p2 (p3 r (ifa (if a (p2 (ifa m (ifa (ifa (ifa (p3 m r (ifa l l)) l) l) l)) m) (p3 l (p2 (ifa m (p2 l m)) (ifa m r)) m)) l) m) l)`
11. `ifa (ifa m (ifa m r)) (ifa (ifa (ifa m l) r) (p3 (p3 l (ifa m r) (p3 r l r)) ( ifa m l) m))`
12. `ifa (p2 m r) (ifa (p2 (ifa (p2 l m) r) m) (p3 l m (ifa r (p3 l (ifa m r) r))))`
13. `p3 (p3 l (ifa m r) r) (ifa (p3 m m (p2 r (ifa (p3 r (ifa m l) (ifa m l)) l))) l ) m`
14. `p3 r (ifa m (p3 l l (ifa (ifa m m) r))) m,89.0,Just 4) (ifa (p3 (ifa m l) r (ifa m l)) (ifa m (p3 (p3 l (ifa m r) m) r (ifa m l)))`
15. `ifa m (p3 (ifa l (p3 r (ifa (ifa l l) (ifa (ifa r r) r)) (ifa r r))) (ifa (ifa m r) r) m)`
16. `p3 (ifa (ifa (ifa m l) l) l) (p3 m (ifa (ifa (ifa (p2 m l) l) (ifa r r)) l) ( ifa m (ifa m r))) r`
17. `ifa (ifa (ifa m r) r) (p3 r (ifa (ifa m (p3 (ifa m (ifa m r)) (ifa (ifa r l) r) (ifa m l))) (ifa l (p3 (ifa m l) (ifa m l) (ifa m (ifa m r))))) m)`
18. `p3 (p3 m r (ifa (p2 m m) l)) l (ifa m r)`

Let us also examine average term sizes of the Simple Simbolic Regression problem, again based on 10 runs.



In this case the difference is not that radical, but still we can see that runs of SSR using ramped half-and-half tend to have terms approximately twice as big as runs using geometric strategy.

5.3 Even-parity problem

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*.

The reason why examine this problem is that it has been used by many researchers as a benchmark for GP [4].

We compare our results with that obtained by Yu in [4], where is presented approach to evolve recursive and modular programs by use of higher-order functions and λ-abstractions. Since the results are highly sensitive to set of *building blocks*, we use very similar set of *building blocks* in order to make the results comparable.

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

The type $[Bool]$ stands for *list of Bools* and for purpose of this problem is considered atomic.

Unlike in [4], 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.

The `fold` function is higher order function widely used in *Haskell* programming language, recursively defined as follows.

```
foldr ::  (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Let us demonstrate `foldr`'s behavior on the following simple example.

```
foldr xor False [True,False,True]
= xor True (foldr False [False,True])
= xor True ( xor False ( foldr False [True] ) )
= xor True ( xor False ( xor True (foldr False [] ) )
= xor True ( xor False ( xor True False) )
= False
```

The reasons why is this difficult problem for GP to solve are following [4]:

- Since a single change of one input generates a different output, the output is highly sensitive to all inputs.
- The building blocks set does not contain functions *xor* or *eq*, which are keys to solving the problem. But those two functions may be discover by GP.

We use the same fitness function as in [4]. The fitness function examines the individual by giving it all possible boolean lists of length 2 and 3. Thus resulting in $2^2 + 2^3 = 12$ test cases. For each correct response it receives $1/12$, therefore correct solution receives fitness value 1.

It is interesting that such a few fitness cases are sufficient to find general solution (it was so for every solution I tried). It comes from the Γ, the solution tends always be `foldr` with *xor* defined by the lambda abstraction.

5.3.1 Experiments

Series of five experiments have been performed to test properties of our system. Each experiment consisted of 50 independent runs of GP algorithm. Each run had maximally 50 generations and 500 individuals as population size.

In those experiments we examined influences of the following factors:

- Option of preserving the best individual into the next population.
- Option of performing η-normalization of generated individuals (conversion from *lnf* to $\beta\eta$-*nf*, see 2.5.1).
- Representation of individuals: sexpr-tree vs @-tree.
- Option of performing optimized abstraction elimination (see 3.3.1) instead of the simple one.

The setup of the first experiment was following:

- *Geometric* search strategy for generating individuals with default $q = 0.75$. This option is unchanged for all five experiments.
- No best individual preservation.
- η-normalization was not performed.
- Individuals represented as sexpr-trees.
- Simple abstraction elimination.

Thus the simplest one setup; all extra features are off. In each following experiment one extra feature will be turned on, in hope that it will enhance the performance.

The correct individual was yield in 17 out of 50 runs, i.e., 34% success rate.

Interestingly, in 2 runs out of 50, a correct solution was present in the initial population.

The smallest value in this graph is used to indicate the minimum effort (i.e. number of individuals to be processed) for GP to yield a correct solution.

Thus for this experiment this analysis suggest to perform only term generating and no evolution at all, because the smallest value 56500 is in generation 0.

This experiment took 59 minutes (on average desktop computer).

Second experiment was different from the first one in that it preserved the best individual to the next generation.

It scored 22/50, i.e., 40% success rate. A correct solution was created once in the initial generation and again it was sufficient to make the value in generation 0 be the smallest value in the $I(M, i, z)$ curve; now 114000 individuals to be processed. This experiment also took 59 minutes.

Third experiment was different from the second one in that η-normalization of generated individuals was performed.

It scored 27/50, i.e., 54% success rate. Two runs contained a correct solution in the initial population. But now the $I(M, i, z)$ was best for generation 1 with 56000 individuals to be processed. This experiment took 53 minutes.

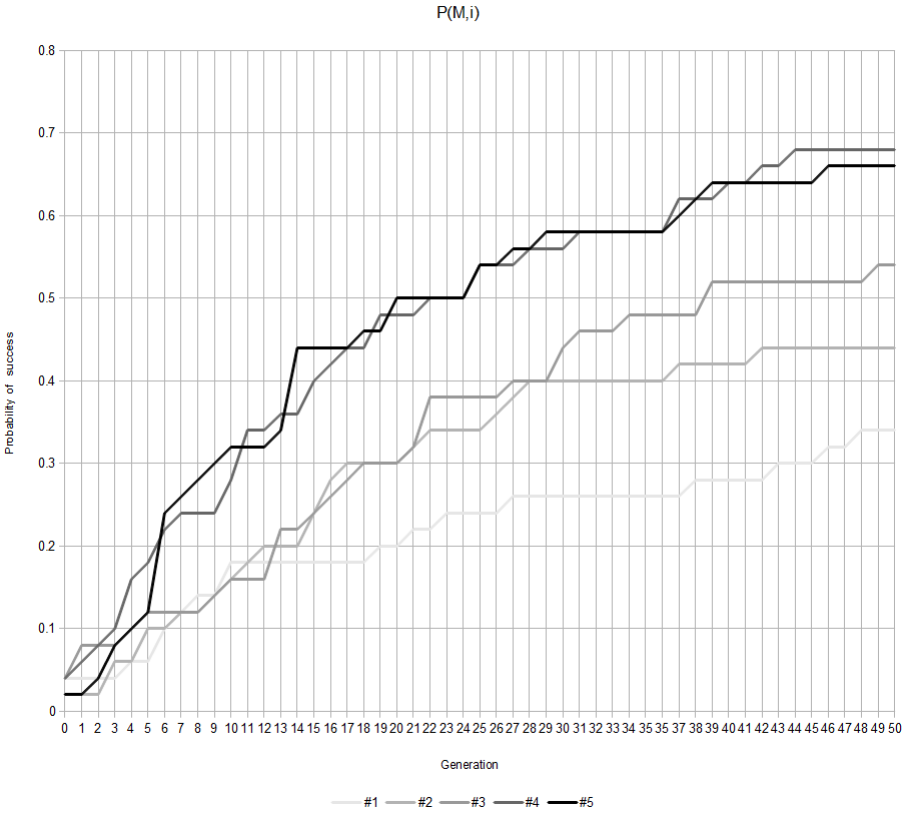
Fourth experiment was different from the third one in that @-tree representation was used instead of sexpr-tree.

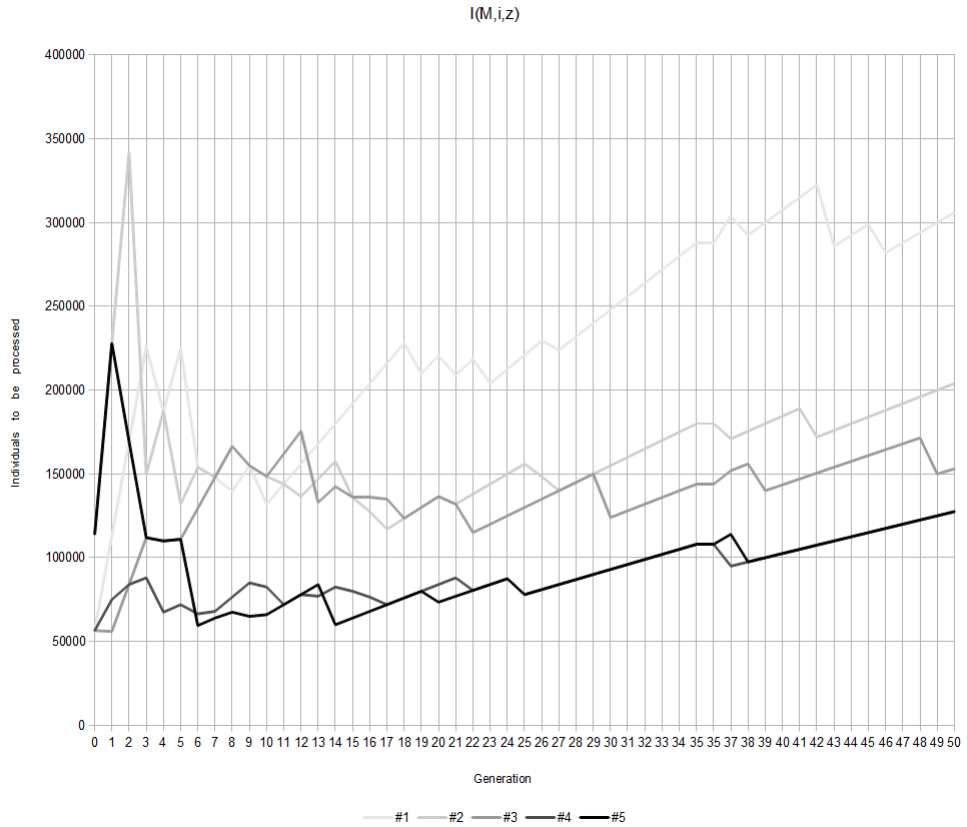
It scored 34/50, i.e., 68% success rate. Again two runs contained a correct solution in the initial population. The $I(M, i, z)$ was best for generation 0 with 56500 individuals to be processed. This experiment took 64 minutes.

As we will see, this setup was the most successful one.

Fifth, last experiment was different from the fourth one in that optimized abstraction elimination was used instead of the simple one.

It scored 33/50, i.e., 66% success rate. One run contained a correct solution in the initial population. The $I(M, i, z)$ was best for generation 6 with 59500 individuals to be processed. This experiment took only 30 minutes.


P(M,i)
Probability of success
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Generation
#1
#2
#3
#4
#5


I(M,i,z)
Individuals to be processed
0
50000
100000
150000
200000
250000
300000
350000
400000
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 35 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Generation
#1
#2
#3
#4
#5

5.3.2 Comparison with other results

Our results are less successful then those presented by Yu [4]; they scored 40/50, i.e., 80% success rate. Her $I(M, i, z)$ was best for generation 4 with 17500 individuals to be processed. (We scored 34/50 (68%); min $I(M, i, z) = 56,500$.)

Yu also uses crossover as the only genetic operator, thus we can narrow the discussion to discussion of generating method and crossover method.

Design of our crossover comes from that we wanted it to be very simple generalization of standard tree-swapping crossover. Whereas the crossover used by Yu is more structure oriented — as is stated in [4] the crossover *can only operate between nodes in two main programs or between nodes in the same kind of λ abstraction (i.e., λ abstractions that represent the same function arguments to the same higher order function).* And also variable terminals of Yu's lambda functions are constrained to be only the variables of that lambda function, not from some outer scope. This may be advantage for this particular problem because xor does not need any outer variable (recall that most common solution is combination of xor and foldr in some similar way as foldr exaple above). From an optimist's point of view 34/50 versus 40/50 is not such a huge difference when we take to account that our general tree-swapping crossover is competing with specialized one.

Since our method has better success rate for generation 0, it would be interesting to try combination of our generating method with crossover used by Yu.

Another optimistic observation is that our result at least outperforms other results mentioned in [4] with which Yu's method is compared.

Generic genetic programming scored 17/60 (28%); min $I(M, i, z) = 220,000$.

GP with ADFs scored 10/29 (34%); min $I(M, i, z) = 1,440,000$.

5.4 Flies and Apples

Flies and Apples is problem consisting in breeding control programs for a fly agent in a simple world.

Whereas previous problem is a kind of problem where we know precisely what problem we are solving and where we are more interested in performance statistics than in the solutions since we know the correct solution in advance, problem we are going to present now is of more playful nature. In this problem we are not interested in numerical results, instead we are hoping for interesting *behavior* to emerge.

We start with defining a simple simulation and goal for agents. Then we prepare first testing simulation world. After that we write some agent programs by hand. After that we identify some useful recurring themes occurring in those programs and pack them into functions which will become building blocks. During this process are also identified useful sensory data to be prepared for the agent program. After that we run the evolution and iteratively continue in this process — we may add more testing worlds, add more functions or change them, etc.

Since we do not need to satisfy the *closure* constraint, it is much easier to perform such a "human-computer jam" than it would be in standard GP. And it is further simplified by use of functional language (such as *Haskell*) where almost every construct can be expressed as function (or value).

Another advantage of typed GP over standard GP is that it is more capable of handling huge set of building blocks, since the search space can be significantly smaller due to fact that types must match.

Fly world is a square grid. Each cell is ether empty or contains precisely one object. There are three kinds of objects:

- Wall
- Apple
- Fly

Apples and flies have *energy*.

Moreover, each fly has its control program and inner state, so called *registers*.

There is a queue of all flies in the fly world determining which fly is next to make a move. After the fly in the front of the queue performs its move, it is put to the back. In other words, flies are cyclically taking turns similarly like in common card games.

A control program of a fly has as input a collection of various sensory information and returns as output agents move and agents registers for the next turn.

Fly can perform two kinds of moves:

- *Trave.* Moves the fly to one of four adjacent cells determined by direction specified in the move. Result of the travel move depends on the content of the target cell.

  - If it is empty, then the fly simply moves to that cell.
  - If it contains wall, then no movement is performed.
  - If it contains an apple, then the fly moves to that cell, the apple is eaten and its energy is added to energy of the fly.
  - If it contains another fly, then the fly moves to that cell, the fly with bigger energy stays alive and the weaker one is eaten and its energy is added to energy of the stronger fly.
- *Split.* Puts a daughter fly to one of four adjacent cells. Together with the direction, amount of energy and new registers for the fly are specified by this move. The mother gives this energy to its child. In order to be successful, the target cell must be empty and the mother fly must have energy greater then 1.

Agent's fitness is computed as sum of fitness cases; each fitness case consists of one fly world and fitness in a fly world is computed as sum of agent's energy and of energy of all his descendants (daughters, granddaughter, ...) after predefined number of rounds.

Let us look in more detail on input for fly. It is collection of various useful information. Its components are:

- Current value of fly's Registers.
- Energy of the fly.
- Direction of the last successful travel move.
- Boolean value indicating whether the last move was successful.
- Direction of the nearest apple.
- Distance to the nearest apple.
- Energy of the nearest apple.
- Direction of the nearest fly.
- Distance to the nearest fly.
- Energy of the nearest fly.
- Direction of the "center of gravity" of all apples.
- Distance to the "center of gravity" of all apples.

Set of building blocks is following.

$$\begin{aligned}\sigma &= Input \rightarrow Output \\ \Gamma = \{&dUp : Direction, \\ &dDown : Direction, \\ &dLeft : Direction, \\ &dRight : Direction, \\ &output : Move \rightarrow Registers \rightarrow Output,\end{aligned}$$

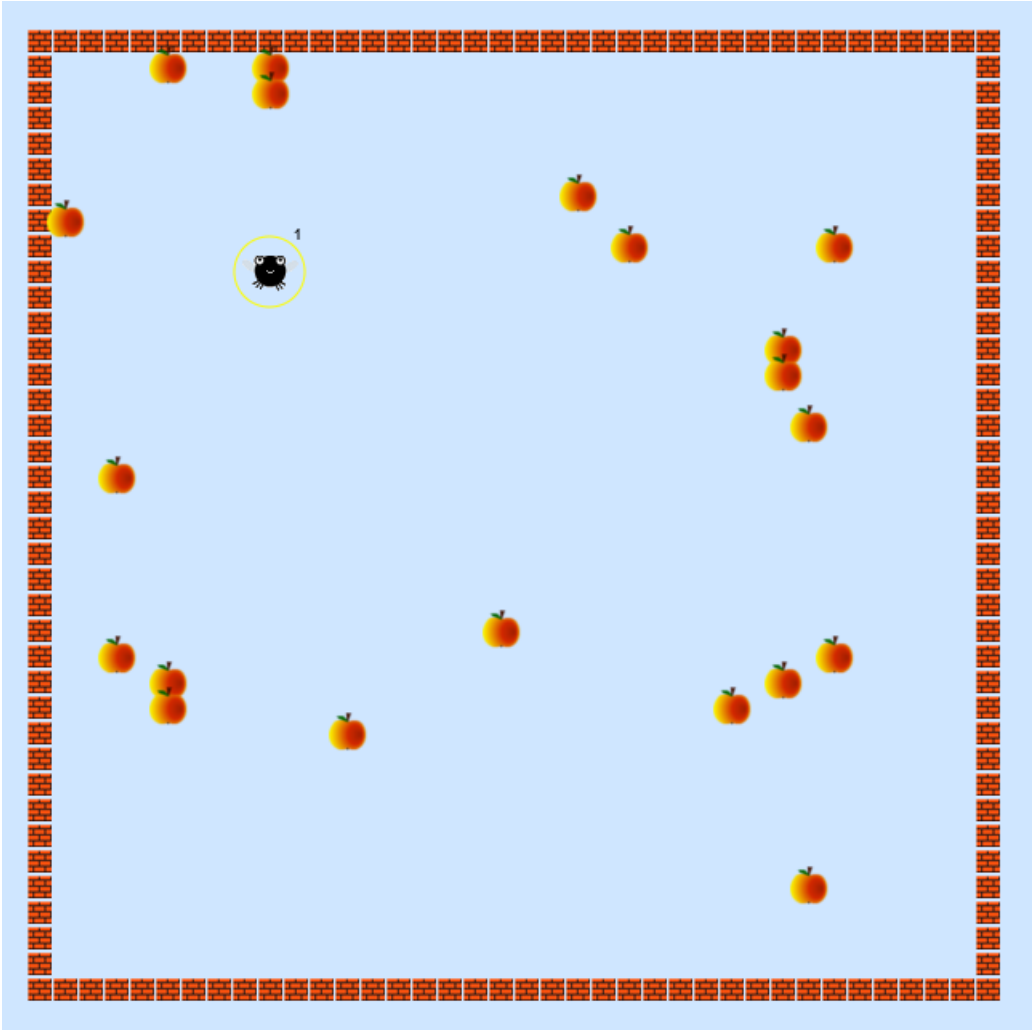
$$
\begin{aligned}
travel &: Direction \to Move,\\
split &: Direction \to Int \to Registers \to Move,\\
easySplit &: Input \to Move,\\
myEnergy &: Input \to Int,\\
myLastTravel &: Input \to Direction,\\
myWasSuccess &: Input \to Bool,\\
nAppleDir &: Input \to Direction,\\
nAppleDist &: Input \to Distance,\\
nAppleEnergy &: Input \to Int,\\
nFlyDir &: Input \to Direction,\\
nFlyDist &: Input \to Distance,\\
nFlyEnergy &: Input \to Int,\\
cAppleDir &: Input \to Direction,\\
cAppleDist &: Input \to Distance,\\
myRegs &: Input \to Registers,\\
xGet &: Input \to Int,\\
yGet &: Input \to Int,\\
zGet &: Input \to Int,\\
dGet &: Input \to Direction,\\
xSet &: Int \to Registers \to Registers,\\
ySet &: Int \to Registers \to Registers,\\
zSet &: Int \to Registers \to Registers,\\
dSet &: Direction \to Registers \to Registers,\\
xInc &: Registers \to Registers,\\
yInc &: Registers \to Registers,\\
zInc &: Registers \to Registers,\\
rotCW &: Direction \to Direction,\\
(==) &: Int \to Int \to Bool,\\
(<=) &: Int \to Int \to Bool,\\
if' &: Bool \to Output \to Output \to Output,\\
if' &: Bool \to Move \to Move \to Move,\\
if' &: Bool \to Direction \to Direction \to Direction,\\
if' &: Bool \to Int \to Int \to Int,\\
if' &: Bool \to Distance \to Distance \to Distance,\\
if' &: Bool \to Registers \to Registers \to Registers,\\
0 &: Int,\\
1 &: Int,\\
2 &: Int\}
\end{aligned}
$$

We will not discuss it in great detail, only briefly: First four elements are four direction values. Next four functions are constructors for $Output$ and $Move$ type. Next twelve functions are for accessing components of $Input$. Next eleven functions are related to $Registers$. The $rotCW$ performs clock-wise rotation of a direction. Next two functions are comparison functions. Next six are various ifs. And last three are integer constants.

This gives 44 elements of building block context. It is quiet huge context, especially if it was a standard GP $T \cup F$.

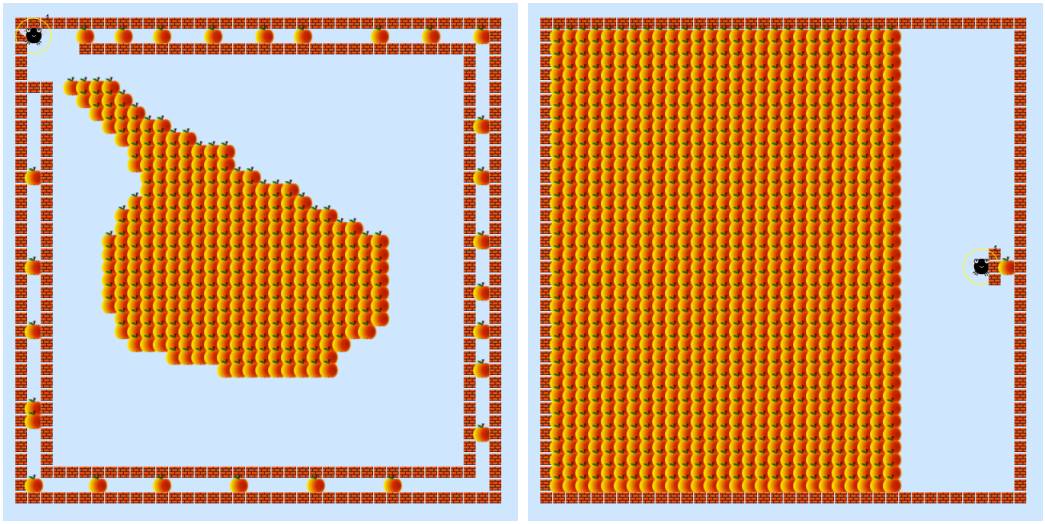
So the set of atomic types is following =
$\{Move, Bool, Registers, Input, Direction, Int, Output, Distance\}$.

We currently use three worlds. First one looks like this:



Other two worlds are reaction on typical behavior of a trivial and boring fly which was successful enough to prevent further evolution. This boring strategy was just following the path of the nearest apple.

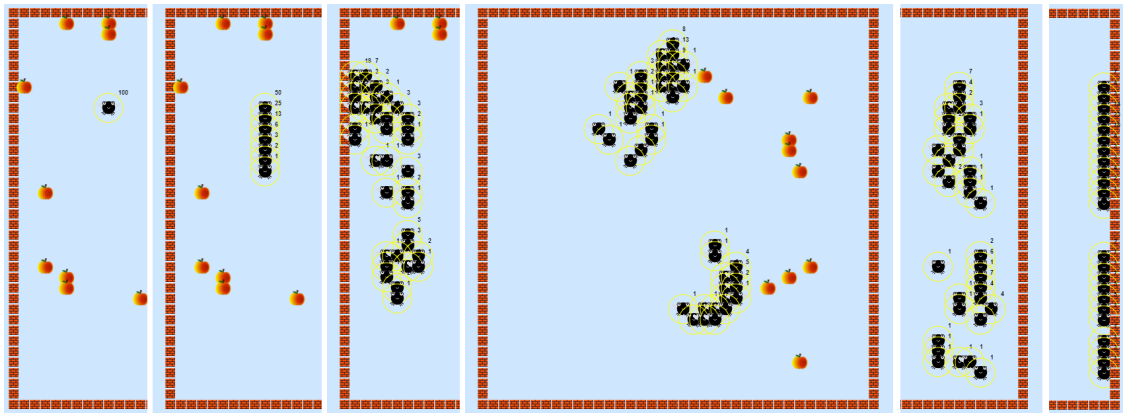
Following two worlds are therefore "Devil's gift" for this fly.



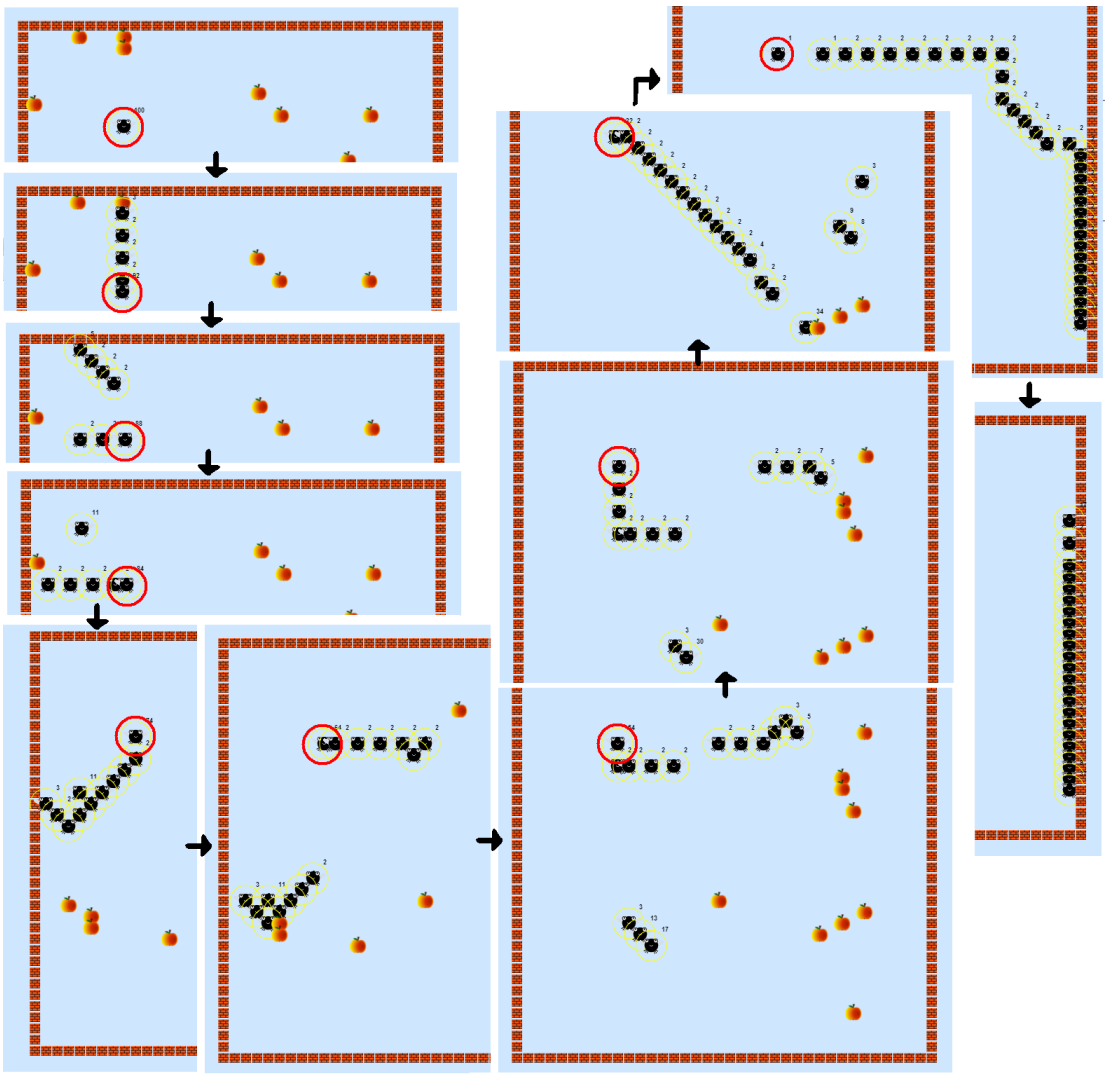
In the second world the boring fly enters the corridor and misses the mega apple. The third world escalates previous idea into absurdity. The boring fly eats nothing since it dumbly tries to eat the apple behind the wall.

By introduction of those two new worlds new kinds of behavior emerged, but not interesting enough to be worth explaining here. One can, however, check those behaviors out in the gallery of interesting fly behaviors, included on the enclosed CD.

In the initial experiments there was problem with that flies did not want to split. This was due to that they were given default energy 1. Since this problem is not tied by some unchangeable definition, we changed this default value to 100 and aded a siplified version of split constructor. That had aftermath in new kinds of behavior.



Application of this kind of approach is for example computer games design where non-linearity and adaptability to players actions are highly appreciated. Also games are not tight by some fixed specification so this approach might be very appropriate in this industry.



6. Previous related work

6.1 Evolution of Constrained Syntactic Structures

Evolution of Constrained Syntactic Structures is mechanism proposed by Koza in [1] for relaxing the *closure* requirement.

Problem specific *rules of construction* are used to constrain which function node can have which child node.

Those rules are used in generating method similar to the standard one.

For purposes of crossover types of symbols are listed - e.g. in the problem of *symbolic multiple regression* (and various other problems) those types are

- the root, and
- non root points.

Crossover is performed by first selecting any node in the first parent and after that by selecting node of the same type as the first node.

System based on types do this task more indirectly and in a simple systematic manner.

6.2 Strongly Typed Genetic Programming

One of first such type based systems was that described by Montana in [5]. This system is called Strongly Typed Genetic Programming (STGP). Beside simple types he also uses kind of polymorphism called *generic functions* based on table lookup.

Used generating method is generalization of the standard one dealing with type restrictions and . Moreover, checking preventing selection of node which makes it impossible to complete the subtree is performed.

The crossover works in similar manner as that Koza used in his Constrained Syntactic Structures. Such a crossover is less cautious then that used in our system since it may fail to find a node with desired type in the second parent. If the crossover fails, then the crossover returns either parents or nothing.

We were briefly measuring fail rates for such a crossover and it was somewhere around 15% for even-parity problem. Thus we prefer to perform the more cautious version described in 3.3.2.

This system does not involve use of lambda abstractions or higher-order functions.

6.3 Higher-Order Functions and Lambda Abstractions

In [4] Yu presents a GP system utilizing polymorphic higher-order functions [1] and lambda abstractions.

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.

Approach described in this work is more specialized then our approach in the following ways.

- 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.
- Whereas we see lambda head of lambda function as another kind of node (as parent node of the body of the function), in this work whole lambda function is considered more as single point with internal structure. Higher-order functions and its lambda abstractions are said to form *two-layer-hierarchy* called in this work *structure abstraction.*
- The crossover used in this work *can only operate between nodes in the two main programs or between nodes in two lambda abstractions that represent the same function arguments to the same higher order function.*

The difference between approach presented in this work and our approach may be stated in the following way. Approach presented in [4] tries to adopt some useful constructs from domain of functional programming and use them for a benefit in problem solving by GP. Whereas our approach is to perform genetic programming directly in the simply typed lambda calculus.

6.4 Functional Genetic Programming with Combinators

In [8] by Briggs and O'Neill is presented technique utilizing GP with combinators in similar way as we do.

The difference between approach presented in this work and our approach is that in this work terms are generated straight from *library* (their term used for context) of combinators, whereas we first generate λ-terms which are subsequently translated into combinator terms.

Thus we can in our system see the combinator representation as option which can be turned off, whereas their system has this feature "hard-wired". On the other hand, their advantage is that direct generation of terms generates shorter combinator terms than indirect generation.

They are using more general type system then us — the Hindley–Milner type system. So the system is polymorphic.

They also discuss the properties of exhaustive enumeration of terms and compare it with GP search. Since our successful term generating *geometric* strategy is

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

based on undisciplined exhaustive enumeration, this is another interesting thing that connects our two systems.

Another really interesting concept is their *Generalized genetic operator* based on term generation. Suppose we have some terms we want to operate on (for one term it corresponds to mutation, for two (parent) terms it is crossover, etc.). In order to do that we construct *library* of all subterms of all terms we want to operate on and from this library we generate new individuals by supplying it with desired type.

It would be nice to use some ideas inspired from this work in the future improvements of our system.

7. Future work

7.1 Polymorphic variant

At this point I have almost complete implementation of polymorphic generalization of the system.

This involves extending the type system so that we also have type variables.

Lets very briefly sketch the key phases.

Then instead of simple equality checking of the current type with return type of the function, we try to fined *most general unification* of those two type terms. If it succeeds, than we may use this function symbol (or constant). Important part is to deal in the right way wit obtained substitution.

7.2 Mutation

Since we have term generating method, it is easy to define subtree changing mutation.

Also node changing mutations are easy.

It would be also nice to perform some kind of informed (or clever) mutation - that is mutation that tries more possibilities and selects the best one (i.e. performs a local search).

In the lambda calculus context, it would be interesting to try some reduction based mutations — simplifying the term by bringing it to normal form form which it was dragged away by crossovers and other mutations.

Also mutations of number-like constants would be useful.

7.3 Big Context

Big Context is an unfinished problem involving breeding of clasical functional functions as `head`,`tail`,`map`,`elem`,`filter`. We first take small sufficient Γ for each function to be evolved. After that we fuse those contexts into one big context and check whether the functions are still evolvable. Or we may try to evolve several functions at once by evolving tuples of them.

We have postponed this problem since it is much more suitable for polymorphic system.

7.4 Parallelisation

Since Haskell is very good at Parallelisation and GP is easily parallelised, it comes naturally to connect the two.

7.5 Future ideas with Inhabitation trees

I have lot of ideas involving inhabitation trees and informed term generation. Number labeled inhabitation tree may be used as more informed version of geo-

metric strategy. Variation of ant colony optimization could be used as vehicle for this informed term generation. This ACO would differ from classical one in that it would use trees instead of paths. Ant may select one edge (or-node) or clone himself (and-node). The global criterion determining the pheromone trail intensity are based upon fitness and local heuristic would be some term shortening factor. [1]

[1] Unfortunately I have run out of time for writing this thesis so I must let this favorite theme of mine mostly unexplained.

Conclusion

The goal of this thesis was to design and implement a system performing GP over some typed functional programming language. As this typed functional programming language we have chosen the *simply typed lambda calculus*. We have also obligated ourselves to do it in a such way that generalizes the standard GP, rather than crates completely new system.

We may say that we have chosen the simplest possible option in both areas — the simply typed calculus and the standard GP. We see this decision as fortunate one — since it enables us add more complex features after sufficient exploration of those simplest cases.

We have started the journey toward this goal by study of subject that intersects both those areas: Method which for desired type and context generates lambda terms.

The way to obtain this method started with inference rules (the defining concepts of the typed lambda calculus), it proceeded through term generating grammars and finished with inhabitation trees.

A* algorithm in combination with inhabitation trees has been utilized to drive systematic enumeration of typed λ-terms in their *lnf*. This enumeration was further parameterized by simple search strategy in order to enable such different approaches as *systematic* generation and *ramped half-and-half* generation to be definable in the means of simple search strategy.

With this apparatus in hands we introduced novel approach to term generation by defining the *geometric* search strategy. This strategy is further parameterized by parameter q, but since we wanted to avoid suspicion that success of this generating method depends on fine-tuning of this parameter, we used its default value $q = 0.75$ in every experiment.

After being generated all terms undergo process of *abstraction elimination*, which enables our simple tree swapping crossover operation which is only genetic operator that we have used in all experiments.

We examined it in three different experiments, which all supported the idea that it has very desirable qualities.

First two experiments, *Simple Symbolic Regression* and *Artificial Ant*, were performed in order to compare the *geometric* strategy with the standard GP term generating method *ramped half-and-half*.

In both experiments were observed improvements in the following aspects.

1. Success rate was improved.
2. Minimal number of individuals needed to be processed in order to yield correct solution with probability 99% was lowered.
3. Run time was significantly reduced.
4. Average size of a term was decreased.

In the case of Artificial Ant problem, all improvements were significant.

These results make me believe that *geometric* strategy might be welcomed reinforcements in the fight against the bogey of the GP community — the *bloat*. Or at least, it seems to work for those two experiments in a such way.

In the third experiment, the *even-parity* problem, geometric strategy showed ability to yield correct solution in the initial generation 8 times out of 200 runs (throughout 5 experiments), which is interesting result since it is uninformed search.

Five consecutive experiments with even-parity problem supported hypothesis that η-normalization of generated terms enhances performance. We have also seen that use @-trees instead of more traditional sexpr-trees as tree representation of individuals is able to enhanced performance.

Use of optimized abstraction elimination instead of its basic variant showed significant improvements in time consumption with little or non effects on performance rates.

Implemented system was designed with importance of interactivity in mind, resulting in server/client architecture for core/GUI components.

Bibliography

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

[2] Koza, J.R., Keane, M., Streeter, M., Mydlowec, W.,Yu, J., Lanza, G. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Springer, 2005. ISBN 978-0-387-26417-2

[3] Riccardo Poli, William B. Langdon, Nicholas F. McPhee *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[4] T. Yu. *Hierachical processing for evolving recursive and modular programs using higher order functions and lambda abstractions*. Genetic Programming and Evolvable Machines, 2(4):345–380, December 2001. ISSN 1389-2576.

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

[6] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright. *Type inheritance in strongly typed genetic programming.* In P. J. Angeline and K. E. Kinnear, Jr., editors, Advances in Genetic Programming 2, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996. ISBN 0-262-01158-1.
URL `http://www.mcs.utulsa.edu/~rogerw/papers/Haynes-hier.pdf`.

[7] J. R. Olsson. *Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search.* Dr scient thesis, University of Oslo, Norway, 1994.

[8] Forrest Briggs, Melissa O'Neill. *Functional Genetic Programming and Exhaustive Program Search with Combinator Expressions.* International Journal of Knowledge-based and Intelligent Engineering Systems, Volume 12 Issue 1, Pages 47-68, January 2008.

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

[10] H. Barendregt , S. Abramsky , D. M. Gabbay , T. S. E. Maibaum. *Lambda Calculi with Types.* Handbook of Logic in Computer Science, 1992.

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

[12] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[13] Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

A. Enclosed CD

The enclosed CD contains implementation source codes, installation instructions and some other material, all described in a greater detail in the `readme.txt` file.