Power of Brute-force Search in Strongly-typed Inductive Functional Programming Automation

Susumu Katayama

University of Miyazaki, Miyazaki 889-2155, Japan,
skata@cs.miyazaki-u.ac.jp

Abstract. A successful case of applying brute-force search to functional programming automation is presented and compared with a conventional genetic programming method. From the information of the type and the property that should be satisfied, this algorithm is able to find automatically the shortest Haskell program using the set of function components (or library) configured beforehand, and there is no need to design the library every time one requests a new functional program.
According to the presented experiments, programs consisted of several function applications can be found within some seconds even if we always use the library designed for general use. In addition, the proposed algorithm can efficiently tell the number of possible functions of given size that are consistent with the given type, and thus can be a tool to evaluate other methods like genetic programming by providing the information of thebaseline performance.

1 Introduction

Strong typing is useful for sound programming because it provides constraints for identifying errors in programs at the compilation time. On the other hand, when using a strongly typed functional language like Haskell[3] one can exploit strong typing to enable random programming, that is, when we forget what function to put to some place in our program, we can just combine functions to obtain a type-consistent program that matches, put it there, and then it is often the case the program works correctly. The main topic of this paper is automation of this trial and error process.

Roughly speaking, the approach to construct a set of functions matching the requested type used in this research is doing the following in the breadth-first manner.

1. construct a set of functions whose return type matches the requested type, and
2. for the type of each argument of each function in the set, if ever, do the same thing recursively.

Without the type constraint, repeating that process until small number of depth causes the number of programs explode. However, as we shall see in Section 5, with type constraints the number of matching functions consisted of several functions is sometimes surprisingly small. Its analogy with the cases where doing search for future moves in playing deterministic board games like chess is interesting: because the number of interesting moves is limited by each situation, brute force search without highly heuristic approach like genetic programming can work well.

Strangely enough, while variations of exhaustive search seem successful in playing deterministic board games, in the recent literature I could not find attempts to apply exhaustive search to synthesis of general functions. As for heuristic methods there are some genetic programming approaches. The ADATE System[6] successfully invents some algorithms by using monomorphic, first-order type system and improving synthesized programs that are correct for some examples and incorrect for others. PolyGP[9] uses polymorphic, higher-order type system like the proposed method, and is discussed in Subsection 5.3 in more detail.

2 Foundations of Functional Programming

This section reviews some important ideas of the foundations of functional programming (e.g. [2]) in short, for readers who are not accustomed with them.

2.1 Lambda Calculus

Lambda calculus is a term rewriting system that is Turing-complete. It is itself a very simple functional language. In other words, functional languages are extensions of lambda calculus with syntactic sugars.

In lambda calculus, each variable is a function and can be passed as an argument. Function applications are expressed by just putting the function left-adjacent to its argument like $x\ y$, where y is applied to function x. They are left associative, i.e., $x\ y\ z$ means $(x\ y)\ z$.

$x\ y\ z$ can also be seen as applying two arguments to binary function x. In fact, the set of functions taking a pair of A and B and returning C is isomorphic to the set of functions taking A and returning a function taking B and returning C. This is how lambda calculus usually deals with n-ary functions, when the theory of lambda calculus itself only deals with unary functions.

Let E is a lambda expression (i.e. an expression in lambda calculus). Then the function taking argument variable x and returning E is written as $\lambda\, x.\ E$. Obtaining a function this way is called *lambda abstraction*. Lambda abstraction is a kind of quantification like $\forall$ and $\exists$. Here E may or may not include x. If x is not included in E, x is called *absent* parameter of $\lambda\, x.\ E$.

$\lambda\, x_1\ x_2\ \ldots\ x_n.\ E$ where E is an expression is a shorthand of $\lambda\, x_1.\ \lambda\, x_2.\ \ldots\ \lambda\, x_n.\ E$. This can be viewed as a function taking $x_1, x_2, \ldots, x_n$ as arguments and returning E.

2.2 Combinators

A closed lambda expression, or lambda expression where all the variables appearing is bounded by lambda abstraction is called a *combinator*. All the exported functions in modules in functional languages are combinators.

Some primitive combinators have their names: e.g. $\mathbf{S} = \lambda\, f\ g\ x.\ f\ x\ (g\ x)$ is called distributer and represents term sharing; $\mathbf{K} = \lambda\, x\ y.\ x$ is called cancellator and represents skipping an argument, and sometimes used in $\mathbf{K}\ E$ form to represent a constant function returning E; $\mathbf{I} = \mathbf{SKK} = \lambda\, x.\ x$ represents an identity function.

The set of functions constructed with function applications of $\mathbf{S}$'s and $\mathbf{K}$'s is Turing-complete. In other words, any computable recursive function can be constructed only with combinations of function applications. This is why we synthesize programs with only function applications in inductive functional programming automation algorithms including the proposed one and genetic programming.

2.3 Typed Lambda Calculus

So far I wrote about type-free lambda calculus and combinatory logic, but most modern functional languages are typed, and are based on typed lambda calculus.

In typed lambda calculus each expression is assigned a type. For example, since $\mathbf{K}$ takes an argument with a type, say, a and returns a function that takes an argument with another type b and returns a, the type of $\mathbf{K}$ is $\forall\ a\ b.\ a\ \rightarrow\ (b\ \rightarrow\ a)$. In this type a and b here are generic type variables that correspond to template type variables in C++, and $x\ \rightarrow\ y$ means the function type taking x as the argument and returns y. Because $\rightarrow$ is right-associative, $\forall\ a\ b.\ a\ \rightarrow\ (b\ \rightarrow\ a)$ can also be written as $\forall\ a\ b.\ a\ \rightarrow\ b\ \rightarrow\ a$, which intuitively reflects the fact that this function can be interpreted as a binary function taking type a and type b as arguments and returns a.

$E :: t$ means that the type of expression E is t. For example, $\mathbf{K} :: \forall ab.\ a \rightarrow b \rightarrow a$. Also, $\mathbf{S} :: \forall\ a\ b\ c.\ (b\ \rightarrow\ c\ \rightarrow\ a)\ \rightarrow\ (b\ \rightarrow\ c)\ \rightarrow\ b\ \rightarrow\ a$, and $\mathbf{I} :: \forall\ a.\ a\ \rightarrow\ a$.

One important fact in assigning types to lambda expressions and combinators is that some lambda expressions have infinite types which are usually prohibited in most typed functional languages. Especially, fixed point combinators which are used to implement recursions are defined with such prohibited subexpressions, and thus general recursions cannot be implemented only with $\mathbf{S}$ and $\mathbf{K}$ combinators. A common solution to this limitation is to regard a fixed point combinator $\mathit{fix} :: \forall\ a.\ (a\ \rightarrow\ a)\ \rightarrow\ a$ as yet another primitive combinator. This issue is discussed further in Subsection 4.2.

3 Implemented System

This section describes the specification and the implementation of the system, but as for the implementation here I provide only a sketch. The full detail of the implementation is written in [5].

3.1 Specification

The system reads a Haskell source file describing the available function set, which I call the *component library*. This algorithm constructs the shortest program with the given type satisfying the given property by combining functions in the component library. If the type is not provided, it is inferred from the property by using the conventional Hindley-Milner style type inference algorithm (e.g. [4]).

The current version of the algorithm can deal only with Hindley-Milner style type system, although Haskell extends it with ad hoc polymorphism or type classes. Also, in the experiments shown later I decided to prohibit functions in containers (such as list of functions, tuple of a function and something, etc.), because it is quite rare that such

containers are required, and because by prohibiting them the efficiency improved in a great deal. I did that by introducing two kinds of type variables: one that can match functions and the other that cannot, and made the system identify each type variable.

3.2 System Structure

Figure 1 shows the structure of the implemented system.

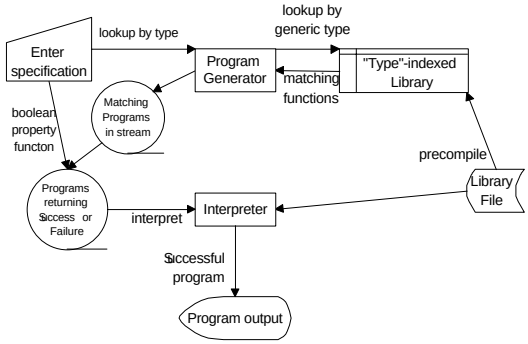

Fig. 1. System structure

The system runs in the following way:

- when the program is invoked, the library is read to the interpreter and the library trie;
- the user-requested type and property are read; if the type is not provided, it is inferred by a conventional algorithm;
- the program generator returns the infinite set of programs that matches the requested type in the form of lazy infinite list (or stream) that is ordered by the program size;
- each generated program is applied to the property function as the argument, and the interpreter runs the resulting term, which is repeated until the return value is true (or success).

3.3 Program Construction

As written in the Introduction, the approach to construct a set of functions matching the requested type used in this research is essentially,

1. to construct a set of functions whose return type matches the requested type, and
2. to do the same thing recursively for the type of each argument of each function in the set, if ever,

in the breadth-first way instead of the depth-first way because the depth is infinite.

This can be achieved in the following way: let X be the infinite set of matching functions defined above, and

1. as a lazy list produce a subset of X that includes programs with size 1, and try each;
2. as a lazy list produce a subset of X that includes programs with size 2, and try each;
3. as a lazy list produce a subset of X that includes programs with size 3, and try each;
   ... and so forth.

Although this may look complicated, it can concisely be written in modern functional languages by using the monad for breadth-first search defined by Spivey[7]. [1]

Here I mention an optimization employed to reduce redundancy in the search space. If there is an expression that includes a subexpression $\mathbf{K}\ E_1\ E_2$, the subexpression can always be reduced to the shorter form E_1, and thus such an expression should always be tried beforehand, when trying shorter programs. The same thing applies to $\mathbf{I}\ E_1$. However, if there is a term sharing things are different, i.e., $\mathbf{S}\ E_1\ E_2\ E_3$ reduces to $E_1\ E_3\ (E_2\ E_3)$, and if E_3 is a long expression this reduction may yield a longer expression. Thus, we obtain the following general optimization rule: *always avoid reducible terms unless the head function duplicates a parameter*.

In the later presentation I write $(\rightarrow)\ a\ b$ instead of $a\ \rightarrow\ b$ if the parameter should not be given to this function when producing programs, or in other words, if supplying the parameter makes the term reducible and the parameter is not shared. The prefixed $(\rightarrow)$ has stronger fixity than the infixed $\rightarrow$.

4 Component Library Design

One easy way of designing the component library is to use $\mathbf{SK}$ plus fix plus constructors and case expressions (or destructors) for each data type. However, this naive way has some tasks that unnecessarily enlarge the search space. This section discusses the policy to design it to avoid unnecessary redundancy in the search space and at the same time to cover large class of functions, based on the results of preliminary experiments.

4.1 Avoiding Absent Parameters

$\mathbf{K} :: \forall\ a\ b.\ a\ \rightarrow\ b\ \rightarrow\ a$ enables absent parameters. On the other hand, if it is used in the component library of the proposed algorithm without any specialization, it causes the search space explode, because its return type a matches any requested type and its argument requests type b, which can match any expression, and thus the algorithm is forced to produce all the expressions without any type constraint as the second argument of $\mathbf{K} :: \forall\ a\ b.\ a\ \rightarrow\ b\ \rightarrow\ a$. Moreover, all the expressions produced in this way are ignored without being used because they are passed as absent parameters, totally wasting the complexity.

One approach to this problem is just to suppress the $\mathbf{K}\ E_1\ E_2$ pattern by defining the type of $\mathbf{K}$ as $\forall\ a\ b.\ a\ \rightarrow\ (\rightarrow)\ b\ a$ and use the optimization written in Section 3.3. What follows shows another approach that avoids using $\mathbf{K}$. This approach prevents

[1] Applying Spivey's monad (stream of finite lists) to my algorithm in a straightforward way causes extreme heap use, because some results of computation remains in the memory for later reuse. In order to avoid that by recomputation, in my implementation I used a function taking an integer and returning a finite list as the monad instead.

simple programs from being filled up with many meaningless combinators, but requires many destructors for each data type.

If a function without any absent parameter is definable from $\mathbf{S}$ and $\mathbf{K}$, it is provably definable without $\mathbf{K}$ by introducing the compositor $\mathbf{B} = \lambda f\ g\ x.\ f\ (g\ x) :: \forall\ a\ b\ c.\ (c \to a) \to (b \to c) \to b \to a$ and the permutator $\mathbf{C} = \lambda f\ x\ y.\ f\ y\ x :: \forall\ a\ b\ c.\ (b \to c \to a) \to c \to b \to a$ as primitive combinators and using $\mathbf{S}$, $\mathbf{B}$, $\mathbf{C}$, and $\mathbf{I}$.

Note that without $\mathbf{K}$ more destructors have to be defined. For example, when the type of natural numbers Nat is defined with 0 and successor function s, if one may use $\mathbf{K}$, only one destructor for Nat is enough, that is, $caseNat$ defined as follows:

$$
\begin{array}{ll}
caseNat & :: \forall\ a.\ a \to (Nat \to a) \to Nat \to a \\
caseNat\ x\ f\ 0 & = x \\
caseNat\ x\ f\ (s\ n) & = f\ n
\end{array}
$$

By using $caseNat$ the function that doubles the argument is defined as $caseNat\ 0\ (\mathbf{B}\ s\ s)$, and the function that returns 0 if the argument is 0 and returns $1\ (=\ s\ 0)$ otherwise can be defined as $caseNat\ 0\ (\mathbf{K}\ (s\ 0))$. However, the latter cannot be defined without $\mathbf{K}$.

If $\mathbf{K}$ may not be used, adding $ifZero$ defined as

$$
\begin{array}{ll}
ifZero & :: \forall\ a.\ a \to a \to Nat \to a \\
ifZero\ x\ y\ 0 & = x \\
ifZero\ x\ y\ (s\ n) & = y
\end{array}
$$

is enough. In general, defining for each argument (Nat, in the above case) of each function argument ($f :: Nat \to a$ in the above case) the version using the argument and that not using it should be enough. However, if there are n parameters this policy requires 2^n destructors, which may be a large amount for some data types. In the experiment in Section 5 I reduced the number by permitting $\mathbf{K}$ with less polymorphic type for some arguments. This issue requires more future discussion, though.

4.2 Fixed Point Combinator and Termination

The fixed point combinator *fix* is defined as $fix\ f\ =\ \mathbf{letrec}\ \{\ x\ =\ f\ x\ \}\ \mathbf{in}\ x$, and has type $\forall a.\ (a \to a) \to a$. It is used to implement general recursions. By applying the identity combinator $\mathbf{I} = \lambda\ x.\ x :: \forall\ a.\ a \to a$ to fix, we obtain an infinite loop $fix\ \mathbf{I} :: \forall\ a.\ a$ which matches any type, increasing the search space tremendously.

Again, one can use $(\to)$ optimization to fix to solve the above problem. However, it is worth discussing whether we should use fix or not. Although a fixed point combinator makes typed lambda calculus Turing-complete, this means the program may not terminate. Thus, if you want to use fix in a search-based programming automation, you have to define timeout in the interpreter. On the other hand, there is a computer language called **Charity** which is designed always to terminate. Although it cannot implement interpreters for usual programming languages, it is still interesting because it has enough ability to implement total functions such as Ackermann's function that are not primitive recursions.

In the experiments in the next section I used paramorphism (e.g. [1]) for each data type instead of fix to obtain terminating programs.

4.3 Resulting Component Library

Based on the discussion so far, I coordinated the component library for the experiments as follows:

— **Primitive combinators:**

$\mathbf{S} \quad :: \forall\ a\ b\ c.\ (b \to c \to a) \to (b \to c) \to b \to a$
$\mathbf{S} \quad = \lambda f\ g\ x.\ f\ x\ (g\ x)$
$\mathbf{B} \quad :: \forall\ a\ b\ c.\ (c \to a) \to (b \to c) \to (\to)\ b\ a$
$\mathbf{B} \quad = \lambda f\ g\ x.\ f\ (g\ x)$
$\mathbf{C} \quad :: \forall\ a\ b\ c.\ (b \to c \to a) \to c \to (\to)\ b\ a$
$\mathbf{C} \quad = \lambda f\ x\ y.\ f\ y\ x$
$\mathbf{I} \quad :: \forall\ a.\ (\to)\ a\ a$
$\mathbf{I} \quad = \lambda\ x.\ x$

— $\mathbf{K}List$ is a version of $\mathbf{K}$ specialized to ignore list parameters.

$\mathbf{K}List \quad :: \forall\ a\ b.\ a \to (\to)\ [b]\ a$ — $[b]$ means list of b's.
$\mathbf{K}List \quad = \lambda\ x\ y.\ x$

— **Natural number constructors:**

$zero \quad :: Int$
$zero \quad = 0$
$successor \quad :: Int \to Int$
$successor \quad = \lambda\ x.\ x + 1$

— **Natural number destructors:**

$paraNat \quad :: \forall\ a.\ a \to (Int \to a \to a) \to Int \to a$
$paraNat\ x\ f\ 0 \quad = x$
$paraNat\ x\ f\ (n+1) \quad = f\ n\ (paraNat\ x\ f\ n)$
$cataNat \quad :: \forall\ a.\ a \to (a \to a) \to (\to)\ Int\ a$
$cataNat\ x\ f\ 0 \quad = x$
$cataNat\ x\ f\ (n+1) \quad = f\ (cataNat\ x\ f\ n)$
$caseNat \quad :: \forall\ a.\ a \to (Int \to a) \to (\to)\ Int\ a$
$caseNat\ x\ f\ 0 \quad = x$
$caseNat\ x\ f\ (n+1) \quad = f\ n$
$ifZero \quad :: \forall\ a.\ a \to a \to (\to)\ Int\ a$
$ifZero\ x\ y\ 0 \quad = x$
$ifZero\ x\ y\ (n+1) \quad = y$
$predecessor \quad :: (\to)\ Int\ Int$
$predecessor\ 0 \quad = 0$
$predecessor\ (n+1) \quad = n$

— **List constructors:**

$nil \quad :: [a]$
$nil \quad = []$ — empty list
$cons \quad :: a \to [a] \to [a]$
$cons \quad = \lambda\ x\ y.\ (x : y)$ — appends an element to a list.

— **List destructors:**

$paraList \quad :: \forall\ a\ b.\ a \to (b \to [b] \to a \to a) \to [b] \to a$
$paraList\ x\ f\ [] \quad = x$

$$
\begin{array}{ll}
paraList\ x\ f\ (a : m) & = f\ a\ m\ (paraList\ x\ f\ m) \\
paraList' & :: \forall\ a\ b.\ a \to ([b] \to a \to a) \to [b] \to a \\
paraList'\ x\ f\ [] & = x \\
paraList'\ x\ f\ (a : m) & = f\ m\ (paraList'\ x\ f\ m) \\
caseList & :: \forall\ a\ b.\ a \to (b \to [b] \to a) \to (\to)\ [b]\ a \\
caseList\ x\ f\ [] & = x \\
caseList\ x\ f\ (a : m) & = f\ a\ m \\
caseList' & :: \forall\ a\ b.\ a \to ([b] \to a) \to (\to)\ [b]\ a \\
caseList'\ x\ f\ [] & = x \\
caseList'\ x\ f\ (a : m) & = f\ m \\
head & :: \forall\ a.\ (\to)\ [a]\ a \qquad \text{— CAR in lisp} \\
head\ (a : m) & = a \\
tail & :: \forall\ a.\ (\to)\ [a]\ [a] \qquad \text{— CDR in lisp} \\
tail\ (a : m) & = m
\end{array}
$$

Note that the same component library is used for all the experiments.

5 Experiments

5.1 Task Description

This section presents results from experiments of composing the following functions:

- $nth :: Int \to [a] \to a$ satisfying
  $nth\ 5\ "widjfgwi" == 'f'$ and $nth\ 1\ "wddidjfgwi" == 'w'$,
- $map :: (b \to a) \to [b] \to [a]$ satisfying
  $f\ (\lambda\ c.\ c ==' c')\ "stock" == [False,\ False,\ False,\ True,\ False]$ and
  $f\ (\lambda\ c.\ c ==' e')\ "peeped" == [False,\ True,\ True,\ False,\ True,\ False]$
- $length :: [a] \to Int$ satisfying $f\ "hageho" == 6$ and $f\ "hoge" == 4$

Correct answers for those tasks are:

$$
\begin{array}{ll}
nth & = \mathbf{B}\ (cataNat\ head\ (\mathbf{C}\ \mathbf{B}\ tail))\ predecessor \\
map & = \mathbf{B}\ (paraList\ nil)\ (\mathbf{B}\ (\mathbf{C}\ (\mathbf{K} list\ cons))) \\
length & = paraList'\ 0\ (\mathbf{K} list\ successor)
\end{array}
$$

In all the tasks the proposed method successfully produced the correct program.

5.2 Results

All the experiments were run on a Pentium4 2.00GHz machine. I used the Glasgow Haskell Compiler ver. 6.2 on Linux 2.4.22, with the `-O` optimization flag.

Table 1 shows the computation time of the proposed method in seconds. Here I provide the minimum and the maximum of three runs. Note that they do not and should not differ in a great deal between runs, because the algorithm is deterministic.

Note that all the experiments finished within seconds. This fact suggests some utility in everyday programming.

It is interesting that the number of type-consistent programs until the correct one is found is within hundreds. Table 2 shows the number of possible programs matching each requested type, ordered by the program size.

Table 1. Computation time (sec.) of the proposed method, and the number of type consistent unsuccessful programs tried

	nth	map	length
min/max of three runs (real)	4.43/4.52	1.59/1.62	0.026/0.044
(user)	4.29/4.37	1.57/1.58	0.010/0.030
# of programs tried until success	619	0	17

Table 2. Number of type-consistent programs for each size.

size	1	2	3	4	5	6	7	8
nth	0	0	2	2	40	113	1027	4626
map	0	0	0	0	0	0	2	7
length	0	1	1	18	29	415	1632	14126

5.3 Comparison with Genetic Programming Approach

PolyGP [9] is a genetic programming algorithm that generates type-consistent Haskell programs in the Hindley-Milner type system. It is a pioneering work that for the first time focused on the Hindley-Milner system to moderately limit the search space in inductive programming automation.

Success rate Unlike my approach presented here, PolyGP is a genetic programming algorithm, and thus it may be unable to find a correct program forever, depending on the initial population. According to [8], 4 of 10 runs for nth and 3 of 10 runs for map are successful. Also, all the successful cases of nth found a correct program within 12000 programs, and those of map found one within 35000. On the other hand, my program always succeeds.

Computation time Because the original code is not efficient, I applied some optimization and ran it in the environment described at the beginning of Subsection 5.2.

The computation time of PolyGP for map task trying 35000 programs spanned from 31.5 sec to 33.7 sec when I tried 10 runs.

Requirements PolyGP requires the user to design the fitness function to reflect how good the program behaves. For nth task this is designed by the difference between the integer argument and the actual position of the returned character, and for map task this is the difference in length and contents between the expected output list and the actual list. Programming those fitness functions can sometimes require more labor than programming the target functions such as nth and map.

Also, experiments of PolyGP in [8] use different library for each task, which my algorithm does not require.

One point we should remember is that using different component libraries between different algorithms can make comparisons unfair, because, to consider an extreme case, if the function that is searched for is included in the library it is easy to find it. On the other hand, we should evaluate algorithms under useful conditions in order to obtain informative results. In Section 4 I tried to show that the library is reasonably selected.

In addition, I used a common library throughout the experiments to show that it is general, which condition is more pragmatic.

6 Conclusions

A system that generates a shortest type-consistent functional program that satisfies the given property by breadth-first exhaustive search is presented and evaluated. Although the tasks might be too easy for evaluation, I guess the system is still useful when, for example, programming overnight and our mind does not work well, provided the computer answers within seconds.

Also, this paper shows that with an appropriate component library the number of type-consistent programs can be surprisingly small. It is interesting that under strongly-typed environment there seems not to have existed brute-force approaches to inductive programming automation in recent literature (and thus with recent CPU power), although there exist heuristic approaches like genetic programming which may be overkill. I do not think exhaustive search methods can universally be applied to synthesis of large programs, but trying the simplest method before inventing heuristic ones and comparing them might be one good research policy.

References

1. Augusteijn, L.: Sorting Morphisms. Advanced Functional Programming, LNCS 1608 (1999) 1–27
2. Field, A. J., Harrison, P. G.: Functional Programming. Addison Wesley (1988)
3. Hudak, P., Fasel, J. H.: A gentle introduction to Haskell. SIGPLAN Notices **27**(5) (1992) T1–T53
4. Jones, M. P.: Typing Haskell in Haskell. Proc. of the 1999 Haskell Workshop (1999)
5. Katayama, S.: Implementing a breadth-first search algorithm for strongly typed inductive functional programming. submitted to The 2004 International Conference on Functional Programming
6. Olsson, R.: Inductive Functional Programming Using Incremental Program Transformation. Research report 189, Doctor scientiarum thesis, University of Oslo, (1994)
7. Spivey, M.: Combinators for breadth-first search. J. Functional Programming **10**(4) (2000) 397–408
8. Yu, T.: Polymorphism and genetic programming. Proc. of Fourth European Conference on Genetic Programming (2001)
9. Yu, T., Clack, C.: PolyGP: A polymorphic genetic programming system in Haskell. Genetic Programming 1998: Proc. of the Third Annual Conference (1998) 416–421