Polymorphism and Genetic Programming

Tina Yu
Chevron Information Technology Company
6001 Bollinger Canyon Road
San Ramon, CA 94583 U. S. A.
tiyu@chevron.com, http://www.addr.com/~tinayu

Abstract. Types have been introduced to Genetic Programming (GP) by researchers with different motivation. We present the concept of types in GP and introduce a typed GP system, PolyGP, that supports polymorphism through the use of three different kinds of type variable. We demonstrate the usefulness of this kind of polymorphism in GP by evolving two polymorphic programs (`nth` and `map`) using the system. Based on the analysis of a series of experimental results, we conclude that this implementation of polymorphism is effective in assisting GP evolutionary search to generate these two programs. PolyGP may enhance the applicability of GP to a new class of problems that are difficult for other polymorphic GP systems to solve.

1 Introduction

Types have been studied and implemented in many modern programming languages, e.g. Haskell, Ada and C++. The ability to support multiple types and to provide type checking for programs has made these languages more expressive and the execution of programs more efficient. Types are counterparts of programming languages.

Genetic Programming (GP) [5] automatically generates computer programs to solve specified problems. It does this by searching through a space of all possible programs for one that is nearly optimal in its ability to solve the given problem. The GP search algorithm is based on the model of natural evolution. In particular, three basic operations (selection, alteration and fitness evaluation) are applied iteratively to a population of programs. Similar to nature where the fittest would survive, the program which best solves the problem would emerge at the end of the GP search process.

The road from untyped to typed GP is led by two goals. Firstly, to enhance the applicability of GP by removing the “closure” requirement. Secondly, to assist GP searching for problem solutions using type information. Koza made the first attempt to introduce types to GP by extending GP with “constrained syntactic structures” when he realized that not all problems have solutions which can be represented in ways that satisfy the closure requirement [5]. Supporters of this argument [9,4] believe that it is important for GP to be able to handle multiple types and advocate excluding type-incorrect programs from the search space to speed up GP’s evolutionary process. Another route to promote the use of types in GP is based on the idea that types provide inductive bias to direct GP search. For example, Wong and Leung included type information in a logic grammar to bias the selection of genetic operation location during program evolution [10]. McPhee *et al.* also showed that a program representation which incorporates *function type* can bias genetic operations and benefit GP search on some problems [8].

These two paths, although with different purposes, are actually interrelated. Types exist in the real world naturally [2]. By allowing problems to be represented in their natural ways, an inductive bias is established which selects solutions based on criteria that reflect experience with similar problems.

In this paper, we first present the concept of types in GP by defining and differentiating *untyped*, *dynamically typed* and *strongly typed* GP. Next, the two different implementations of strongly typed GP, *monomorphic* GP and *polymorphic* GP, are discussed. We introduce our polymorphic GP system (PolyGP) and compare it with other polymorphic GP systems. This system is then used to evolve two polymorphic programs, `nth` and `map`, which represent a class of problems that we believe to be very difficult for other polymorphic GP system to evolve. We analyze a series of experimental results to evaluate the effectiveness of PolyGP polymorphism in assisting GP evolutionary search. Finally, we give our conclusions.

2 Types in Genetic Programming

In its traditional style, GP is not capable of distinguishing different types: the term *untyped* is used to refer to such a system. In the case when the programs manipulate multiple types and contain functions designed to operate on particular types, untyped GP leads to an unnecessarily large search space, which contains both type-correct and type-incorrect programs. To enforce type constraints, two approaches can be used: *dynamically typed* GP and *strongly typed* GP. In dynamically typed GP, type checking is performed at program *evaluation* time. In contrast, strongly typed GP performs type checking at program *generation* time. The computation effort required for these two different type checking methods is implementation dependent. It is not valid to claim that dynamic typing is more efficient than strong typing; nor vice versa. The details of these two type checking approaches are discussed in the following sections.

3 Dynamically Typed GP

Dynamic typing performs type checking when a program is evaluated to determine whether it is a solution, i.e. type-incorrect programs are not allowed to exist in the solution space. However, the search space may contain both type-correct and type-incorrect programs. Consequently, the size of the search space is bigger than that of the solution space (Figure 1).

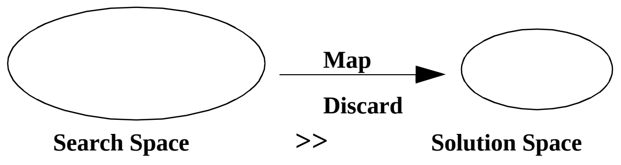

Figure 1 Search Versus Solution Space in Dynamically Typed GP.

The transformation of a type-incorrect into a type-correct program can be implemented using a “legal map” method [11]. For example, a value with an illegal type of “real” can be mapped into a value with legal type “integer”. However, for more complex types such as list or matrix, a proper mapping scheme can be difficult to design. Montana implemented this dynamic typing approach in one of his experiments. When trying to add a 3-vector with a 4×2 matrix, the matrix is considered as an 8-vector, which is converted into a 3-vector by throwing away the last 5 entries. “The problem with such unnatural operations is that, while they may succeed in finding a solution for a particular set of data, they are unlikely to be part of a symbolic expression that can generalize to new data.” [9].

Another way to implement dynamic typing is to discard type-incorrect programs. “The problem with this approach is that it can be terribly inefficient, spending most of its time evaluating programs that turn out to be illegal.” [9]. In his experiments, Montana reported that within 50,000 programs in the initial population, only 20 are type-correct. Because of the complication involved with the mapping and discarding process, dynamic typing is not an ideal way to implement type checking in GP.

4 Strongly Typed GP

Strong typing performs type checking when a program is generated to be a member of a GP population, i.e. the search space is restricted to contain only type-correct programs. This is done through the “legal seeding” method (the initial population is seeded with solutions that do not conflict with the type constraints) and the “legal birth” method (alteration operators are designed such that they cannot generate type-incorrect programs) [11]. Consequently, the size of the search space is the same as that of the solution space (see Figure 2).

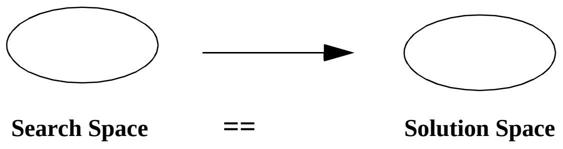

Figure 2 Search Versus Solution Space in Strongly Typed GP.

There are two ways of implementing strong typing in GP: *monomorphic GP* and *polymorphic GP*. Monomorphic GP uses monomorphic functions and terminals to generate monomorphic programs. In contrast, polymorphic GP can generate polymorphic programs using polymorphic functions and terminals. In the first instance, inputs and outputs of the generated program need to be of the specified types. In the latter case, the generated programs can accept inputs and produce outputs of more than one type. Polymorphic GP therefore can potentially generate more general solutions than those produced by monomorphic GP.

4.1 Monomorphic GP Systems

One can perceive Koza’s untyped GP with closure requirement as a monomorphic GP; it is a single-type GP system. The first monomorphic GP system that supports multiple types is Montana’s “basic” STGP [9]. In this system, types are atomic symbols and type compatibility is determined by simply checking whether the two types are the same or not. Later, Haynes *et al* implemented a similar monomorphic GP system (GPengine) to evolve cooperation strategies for a predator-prey pursuit problem [4].

4.2 Polymorphism through Type Variables

In PolyGP, polymorphism is implemented through the use of three different kinds of type variables. *Generic type variables* are used to specify the generated program can accept inputs of any type. For example, type `[G1]` (`G1` is a generic type variable) is used to indicate that the input of `nth` program is a list of values with any type (see Section 5.2). Moreover, to enable the generated program to accept inputs of any type, the functions and terminals used to construct the programs have to be able to handle arguments of multiple types. This is specified using *dummy type variables*. For example, the argument of function `head` is of type `[a]` (`a` is a dummy type variable), which indicates that `head` can take a list of values of any type as input (see Section 5.2).

When a polymorphic function is selected to construct a program tree node, its dummy type variables have to be instantiated to known types. If the type of a dummy variable can't be determined, a *temporary type variable* is used. The temporary type variable can be instantiated to a different type, hence provides polymorphism during program evolution. With a type system that supports these three different kinds of type variables, the PolyGP system is able to generate generic programs while monomorphic GP can not.

On the surface level, it seems that PolyGP is simply a monomorphic GP system plus type variables. Indeed, when type variables are not utilized, PolyGP becomes a monomorphic GP system. However, the use of these type variables can impact the GP search space in ways that are not obvious to GP users. Misuse of these type variables can cause unnecessary overhead to the GP system. In [13], we have provided related information so that users can use PolyGP with or without type variables to suit their target problems.

4.3 Polymorphism through Generic Functions

Generic functions in Montana's STGP system [9] also provide a form of polymorphism. Generic functions are parameterized templates that have to be instantiated with actual values before they can be used to construct program tree nodes. The parameters can be type parameters, function parameters or value parameters. Generic functions with type parameters are polymorphic since the type parameters can be instantiated to many different type values.

However, the implementation of polymorphism in STGP is different from that of PolyGP in the following ways:

- It requires the generation of a type possibilities table;
- It uses a table-lookup mechanism to instantiate type variables;
- It does not provide a systematic way to support function types;
- It does not use *temporary type variables*, hence the system is monomorphic at program evolution time.

The detailed explanation of these differences is provided in Section 3.2.1 of [13].

4.4 Polymorphism through Subtyping

Unlike type variables and generic functions, where polymorphism is for an infinite number of types, subtyping supports polymorphism only for a finite number of types. Moreover, these types are related in a hierarchical manner. For example, a type A can have a subtype B which can have a subtype C. In this case, wherever type A may appear, its subtype B and C may appear. Polymorphism is for the 3 types (A, B, C) only. Haynes *et al.* implemented a polymorphic GP system that supports subtyping for a limited single inheritance type hierarchy (each type is allowed to have a maximum of one subtype and one supertype). They have used the system to solve a maximum clique problem [3].

The Evolutionary Computation Java (ECJ) system by Sean Luke used a set-based approach to support polymorphism: a type can be "compatible" with a set of types. He has used this mechanism to implement subtyping of multiple-inheritance: a type can have a set of subtypes and a set of supertypes. This polymorphic GP system has been used for an internal project at his university [7].

The unstructured nature of a set makes it suitable for implementing polymorphism for a group of types that are not related in a definable manner. Cardelli and Wegner classified this as ad-hoc polymorphism, which includes overloading and coercion [2]. We hope to see work using this kind of polymorphism with the ECJ system.

5 Experiments

To demonstrate the usefulness of the type-variables-polymorphism, we use the PolyGP system to generate two polymorphic programs, `nth` (which returns the n-th element of a list) and `map` (which applies a function to a list). Although simple, these two programs are appropriate because the two programs are generic to *any* type of inputs, which can be handled using generic type variables in our system:

```
nth :: int->[G1]->G1
map :: (G1->G2)->[G1]->[G2]
```

Subtyping GP systems (Section 4.4), which supports polymorphism to a finite number of types, can not be effective with this class of problems Although the generic function STGP system successfully generated the same two programs, it treated the function argument type of the `map` program in an ad-hoc manner. This system is therefore not a general solution to other similar problems. The PolyGP system, on the other hand, provides a systematic way of handling function types. It is therefore capable of dealing with the same class of problems. By generating these two programs, we show that PolyGP can enhance the applicability of GP to a class of problems that are very difficult for other polymorphic GP systems to solve.

Both `nth` and `map` are also recursive programs. To allow GP to evolve recursive programs, a simple method similar to that used in [1] is applied. In this method, the name of the program is included in the function set so that it can be used to construct program trees; hence making recursive calls. However, such implementation may also cause an evolved program to generate an infinite-loop, hence making the evaluation of such a program non-terminating. To handle this kind of error, a maximum number of recursive calls allowed in a program is specified (the length of the input list is used in this case). When this limit is reached, program evaluation halts and returns with a flag to indicate this error. Consequently, no partial credit is given to this kind of program.

Another kind of error that may occur during program evaluation is to apply the function `head` or `tail` to an empty list (which has an undefined value). When this error occurs during program evaluation, a default value is given and the evaluation of the program continues. In this way, partial solutions can be considered for partial credit. For example, a program which is expected to return the fifth element of the list may return the third element of the list due to this error. This program is given partial credit even though a non-optimal program is generated. The implementation details of these two error handling methods are described in [13].

5.1 Experimental Setup

The two kinds of error and their handling methods have complicated our experiments. In particular, it becomes unclear whether the ability of the PolyGP system to generate the two programs should be interpreted as the result of:

1. random sampling of the set of type-correct programs;
2. sampling by genetic operations in the set of type-correct programs that satisfy the two error constraints;
3. genetic operations guided by the fitness function toward a correct solution.

To allow us to make a determination, we have designed and implemented the following four experiments:

1. Programs are generated randomly, no genetic operation is applied.
2. Programs are generated randomly for the initial population. After that, new programs are generated by applying genetic operations on randomly selected programs.
3. Programs are generated randomly for the initial population. After that, new programs are generated by applying genetic operations on programs which are selected based on how well they satisfy the two error constraints. The correctness of the program outputs is not considered.
4. Programs are generated randomly for the initial population. After that, new programs are generated by applying genetic operations on programs which are selected based on a fitness function which evaluates both the correctness of the outputs and the satisfaction of the two error constraints.

The first experiment is conducted by random generation of 100,000 programs. We used the FULL method described in [5], where all program trees are grown into the maximum allowable depth. It has been argued that for most problems, the distribution of solutions in the search space is independent of the program length [6]. We hope this uniform program length tree generation approach will produce an unbiased performance.

For each of the second, third and fourth experiments, 10 runs are performed. This seems to be too small a sample size to be statistically meaningful. However, given that we used the same 10 random seeds for the three sets of experiments, we hope the results reflect more closely the different implementations of the experiments. To trace the performance of the entire run, each run continues until the end, even if a solution has been found.

In each set of the four experiments, the following data are recorded:

- The number of correct solutions found (first experiment) and the number of successful runs (other 3 experiments);
- The mean value of each component of the fitness function.

In terms of implementation, the three sets of GP experiments are basically the same except the fitness value used for selection. With the second experiment, a flat fitness (1.0) is given to every generated program (flatFitness). With the third experiment, the two error fitness values are used as the program's fitness for selection (errorFitness). With the fourth experiment, the combination of error fitness and output fitness is used as the program's fitness for selection (fullFitness).

The GP system uses a steady-state replacement with rank selection. In particular, programs in the population are ranked and the probability of a program to be selected is less than that of the next better program in the rank, based on the "parent scalar" parameter provided. The details of the implementation are described in [12].

5.2 The NTH Program

Problem Description: The `nth` program takes two arguments, an integer `N` and a list `L`. It returns the `N`th element of `L`. If the value of `N` is less than 1, it returns the first element of `L`. If the value of `N` is greater than the length of `L`, it returns the last element of `L`.

Input Types: Input `N` has type `int`; input `L` has type `[G1]`.

Output Type: The output has type `G1`.

Terminal Set: {

```
            L:: [G1], N:: int, one:: int}
```

Function Set: {

```
            if-then-else :: bool->a->a->a,
            head :: [a]->a, tail :: [a]->[a],
            length :: [a]->int, gtr :: int->int->bool,
            less-eq :: int->int->bool,
            minus :: int->int->int,
            nth :: int->[G1]->G1}
```

Genetic Parameters: The population size is 3,000; parent scalar is 0.9965; maximum tree depth is 5; and crossover rate is 100%. Each run continues until 33,000 programs are generated (3,000 are created randomly and 30,000 are created by crossover).

5.2.1 Test Cases

Twelve test cases were used to evaluate the generated programs. Each test case gave `N` a different value from 0 to 11. The value `L`, however, is the same for all 12 test cases; it is a list containing the characters `a` to `j`. Table 1 lists the 12 test cases and their expected outputs.

5.2.2 Fitness Function

A generated program is evaluated with each of the 12 test cases, one at a time. The produced output is compared with the expected output to compute the fitness value for each test case according to equation 1:

Table 1: The 12 test cases for evolving the `NTH` program.

Case	N	L	output	Case	N	L	output
1	0	[a,b,c,d,e,f,g,h,i,j]	a	7	6	[a,b,c,d,e,f,g,h,i,j]	f
2	1	[a,b,c,d,e,f,g,h,i,j]	a	8	7	[a,b,c,d,e,f,g,h,i,j]	g
3	2	[a,b,c,d,e,f,g,h,i,j]	b	9	8	[a,b,c,d,e,f,g,h,i,j]	h
4	3	[a,b,c,d,e,f,g,h,i,j]	c	10	9	[a,b,c,d,e,f,g,h,i,j]	i
5	4	[a,b,c,d,e,f,g,h,i,j]	d	11	10	[a,b,c,d,e,f,g,h,i,j]	j
6	5	[a,b,c,d,e,f,g,h,i,j]	e	12	11	[a,b,c,d,e,f,g,h,i,j]	j

$$Fitness = 10 \cdot 2^{-d} - (10 \cdot rtError) - (10 \cdot reError) \tag{1}$$

where *d* is the distance between the position of the expected value and the position of the value returned by the generated program; *rtError* is 1 if there is an empty-list error; and *reError* is 1 if there is a non-terminating recursion error. The value 10 is used to scale up the fitness values for better precision due to computational round up.

The first part of equation 1 indicates that a program which returns the value at the expected position receives a fitness 10. This fitness value decreases as the position of the returned value is farther away from the position of the expected value. If the returned value is not a part of the input list, this fitness is 0. When no error is encountered during program evaluation, a program can receive a maximum fitness value of 10 for each test case. The fitness of a program is the summation of the fitness for all 12 test cases, i.e. the maximum fitness of a program is 120.

5.2.3 Results

The first experiment, which randomly generates 100,000 type-correct programs, could not find a correct `nth` program. The average fitness is -121.266 where 7.342464 is the average output fitness; -41.9595 is the average empty-list error fitness and -86.6488 is the average recursion error fitness.

For the other three sets of experiments, only the one which applied genetic operations with full fitness (the fourth experiment) found correct `nth` programs. Within 10 runs, 4 of them found a solution. The probability of success is 40%. Moreover, all the successful runs found a solution before 12,000 programs were processed. The shortest program is given as the following. It's interesting to see the system uses (minus N N) to construct value 0.

```
if-then-else (less-eq (length (tail L)) (minus N N))
             (head L)
             (if-then-else (gtr (minus N N) (minus one N))
                    (nth (minus N one) (tail L))
                    (head L)
```

5.2.4 Analysis

The fitness function in equation 1 specifies that an optimal solution not only has to produce the desired output but also has to satisfy two different constraints: a non-terminating error constraint and an empty-list error constraint. To meet the three criteria, the evolutionary process is directed toward different directions. Consequently, the seesaw of evolutionary pressure can favor either of the criteria.

In the fitness function, these three criteria are given equal importance (10 points each). However, the two error constraints are handled in ways which implicitly generated more weight for them. These effects can be illustrated in Table 2 where four categories are defined for a generated `nth` program when evaluated with a test case.

Table 2: The 4 categories of the `NTH` programs.

Program Errors	Output	Penalty	Fitness
non-terminating+empty-list	nothing	20	-20
non-terminating	nothing	10	-10
empty-list	a value	10	partialCredit-10
none	a value	0	calculatedFitness

The analysis shows that programs in the first two categories, which are non-terminating, have the lowest fitness values. Our steady-state implementation, which replaces the worst program in the population only when the new program has better fitness, makes them very unwelcome in the evolutionary process. They soon will be eliminated from the population pool. The evolution then starts to find programs which produce correct output and also satisfy the empty-list error constraint.

Our experimental results are consistent with this analysis. Using the fitness function in equation 1 to direct genetic operations (fullFitness), programs with recursion error were eliminated from the population very fast. After 6,000 programs were processed, the population contained no program with recursion error (see Figure 3(C))[1].

Between processing 6,000 and 8,000 programs, the average output fitness increased (see Figure 3(A)) while the average empty-list error fitness decreased (see Figure (B)), i.e. evolution was driven by the output fitness. After that, both output and empty-list fitness improved consistently. As a result, 4 of the 10 runs found a correct `nth` program.

When genetic operations were guided to evolve programs which only satisfy the two error constraints but not the output correctness (errorFitness), both the average empty-list and the recursion error fitness improve very fast. After 3,000 genetic operations, programs with either of the two errors were eliminated from the population (see Figure 3 (B) & (C)). Unfortunately, this “biased” selection also eliminated potential good programs from the population. As shown in Figure 3 (A), applying genetic oper-

1. With steady-state replacement, there is no concept of “generation”. Hence, data are reported every 3,000 programs (the size of the population) processed.

ations randomly on the set of programs which satisfy the two error constraints does not improve output fitness (due to premature convergence). Consequently, no solution was found at the end of the 10 runs.

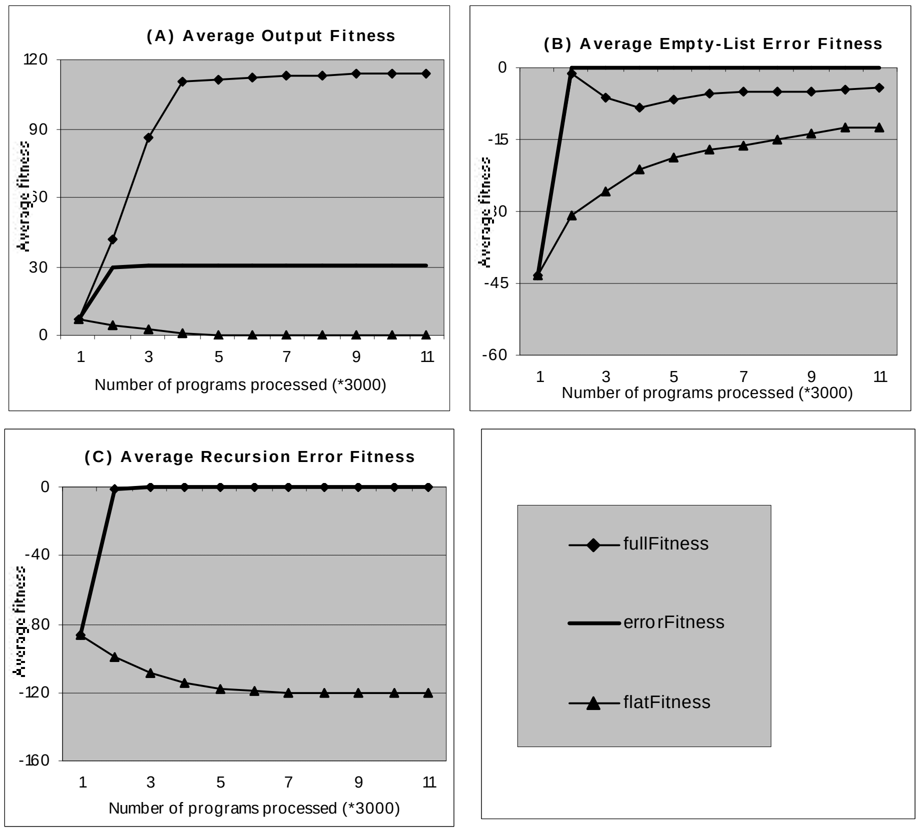

Figure 3 Experimental Results for the NTH program.

Without the guidance of a fitness function (flatFitness), genetic operations did not provide a consistent improvement of the solutions. Although the average empty-list error fitness increased (see Figure 3 (B)), the average output fitness and recursion error fitness decreased as the number of program processed increases (see Figure 3 (A) & (C)). Consequently, no solution was found at the end of the 10 runs.

5.3 The MAP Program

Problem Description: The `map` program takes inputs of two arguments, a function `F` and a list `L`. It returns the list obtained by applying `F` to each element of `L`.

Input Types: The input `F` has type `G1->G2` and the input `L` has type `[G1]`

Output Type: The output has type `[G2]`

Terminal Set: {

```
L::[G1],nil::[a],F::(G1->G2)}
```

Function Set: {

```
if-then-else :: bool->a->a->a,
F :: G1->G2, head :: [a]->a,
tail :: [a]->[a], cons :: a->[a]->[a],
null :: [a]->bool,
map :: (G1->G2)->[G1]->[G2]}
```

Genetic Parameters: The population size is 5,000; parent scalar is 0.999; maximum tree depth is 5; and crossover rate is 100%. Each run terminates when 55,000 programs are generated (5,000 are created randomly and 50,000 are created by crossover).

5.3.1 Test Cases

Two test cases were used to evaluate the generated programs. The two test cases have different values for argument `L`: the first one is a list with 10 elements whose values were the characters `A` to `J` while the second one is an empty list. The function `F`, however, is the same for both test cases. It is a function which converts an alphabetic character into a number, i.e. `A` to `1`, `B` to `2`, `C` to `3` and so on. Table 3 lists these two test cases and their expected outputs.

Table 3: The 2 test cases for evolving the `map` program.

Case	F	L	Expected Output
1	atoi	[A,B,C,D,E,F,G,H,I,J]	[1,2,3,4,5,6,7,8,9,10]
2	atoi	[]	[]

5.3.2 Fitness Function

The fitness of a program is computed based on how close the return list is to the expected list. There are two elements in this criterion: 1) whether the returned list has the correct *length* and 2) whether the returned list has the correct *contents*. Equation 2 is the fitness function used to measure both elements for each test case.

$$Fitness = -2 \cdot \left| length(L_e) - length(L_r) \right| + \sum_{e \in L_e} 10 \cdot (2^{-dist(e, L_r)})$$
$$-(10 + 2 \cdot length(L_e)) \cdot rtError - (10 + 2 \cdot length(L_e)) \cdot reError \qquad (2)$$

where L_e is the expected list and L_r is the list returned by the generated program; $dist(e, L_r)$ is the distance between the position of *e* in the expected list and in the returned list. If *e* does not exist in L_r, i.e. $e \notin L_r$, this value is ∞. The *rtError* is 1 if there is an empty-list error. The *reError* is 1 if there is a non-terminating recursion error. Similar to equation 1, the value 10 is used to scale up the fitness value for better precision.

The first item in equation 2 measures whether the returned list has the same length as the expected list. Each discrepancy is penalized with a value of 2.

The second item in equation 2 measures whether the returned list has the same contents as the expected list. For each element in the expected list, the distance between its position in the expected list and in the returned list is measured. If the element is in the correct position in the returned list, a fitness value of 10 is given. This value decreases as the position of *e* in the returned list is farther away from the expected position. In the case that *e* does not exist in the returned list, this fitness is 0. The same measurement is applied to each element in the expected list. For the first test case, a program which generates a list with the correct length and contents would receive a fitness value of 100. For the second test case, a program which generates a list with the correct length and contents will receive a fitness value of 0.

The third and the fourth items in equation 2 measure whether an empty-list or a non-terminating error is encountered during program evaluation. The penalties of these two types of error are the same. It is proportionate to the length of the expected output list (the same as the length of the input list). This decision is due to the recursion limit being the length of the input list. It seems to be reasonable to penalize these two errors using the same criterion.

When no error is encountered during program evaluation, a program receives a maximum fitness value of 100 for the first test case and 0 for the second test case. The fitness of a program is the summation of the fitness for the two test cases, i.e. the maximum fitness of a program is 100.

5.3.3 Results

Similar to the results of the `nth` experiment, none of the randomly generated 100,000 programs is a correct `map` program. The average fitness is -82.6504 where -19.9724 is the average length fitness; 0.0976 is the average output fitness; -27.9946 is the average empty-list error fitness and -34.7787 is the average recursion error fitness.

Among the three sets of GP experiments, only those which applied genetic operations with the full fitness function found correct `map` programs. There are 3 such successful runs within 10 trials. The probability of success is thus 30%. Moreover, all the successful runs found a correct solution before 35,000 programs were processed. The shortest program is given as the following:

```
if-then-else (null L)
             (head (cons nil nil))
             (cons (F (head L)) (map F (tail L))))
```

5.3.4 Analysis

The experimental results indicate that the `map` program is harder than the `nth` program for PolyGP to generate. Unlike the `nth` program where only one correct return value is required to get the maximum fitness, an optimal `map` program has to be able to process each of the 10 elements in the input list correctly. To meet this objective, one more criterion is added to the fitness function: the length discrepancy between the expected and the generated lists (see equation 2). Moreover, unlike that for the `nth` program, the fitness function for the `map` program gives different importance to the two objectives and the two constraints. In the first test case, 100 points are given to programs which return list with the correct contents; 20 points are given to programs which return lists with the correct length; 30 points are given to programs which produce no non-terminating error and 30 points are given to programs which produce no empty-list error. This is designed to direct GP searching for programs which return lists with the correct contents. In the second test case, the two objectives are not considered in the fitness function while the two constraints are each given 10 points of importance in the fitness function. It is obvious that the purpose of the second test case is to train GP to handle the empty list without producing any errors.

Additionally, the handling methods for the two errors during program evaluation generate another level of evolutionary pressures. Due to such a complicated fitness assignment and constraint handling pressure, the evolutionary process is directed toward many directions. Consequently, the generation of the correct `map` program is harder than the generation of the correct `nth` program.

To analyze the impact of these fitness assignments on the evolutionary process, we identify four categories for generated `map` programs when evaluated with the first test case (Table 4) and the second test case (Table 5)

Table 4: The 4 categories of `map` programs for the first test case.

Program Errors	Output	Error Penalty	Length Penalty	Fitness
non-terminating + empty-list	nothing	60	20	-80
non-terminating	nothing	30	20	-50
empty-list	a list	30	calculated penalty	partial credit - 30 - calculated penalty
none	a list	0	calculated penalty	calculated fitness - calculated penalty

Table 5: The 4 categories of `map` programs for the second test case.

Program Errors	Output	Error Penalty	Length Penalty	Fitness
non-terminating+empty-list	nothing	20	0	-20
non-terminating	nothing	10	0	-10
empty-list	empty list	10	0	-10
none	empty list	0	0	0

For both of the test cases, programs with non-terminating error are heavily discriminated in the population (the first and the second categories, with fitness values -100 and -60 respectively). Consequently, they would be eliminated from the population first. After that, evolution is a process of competition among programs to meet the three other criteria. The experimental results confirm our analysis. Using fitness defined in equation 2 to select programs for reproduction (fullFitness), the GP evolutionary process was dominated by recursion error fitness before 20,000 programs were processed (see Figure 4(D)). During this period of time, both the average output and length fitness also improved (see Figure 4 (A) & (B)). This indicates that programs without recursion error also produce more accurate outputs. The empty-list error fitness, however, did not increase consistently during this period of time. This result can be explained by the see-saw effect described in [11].

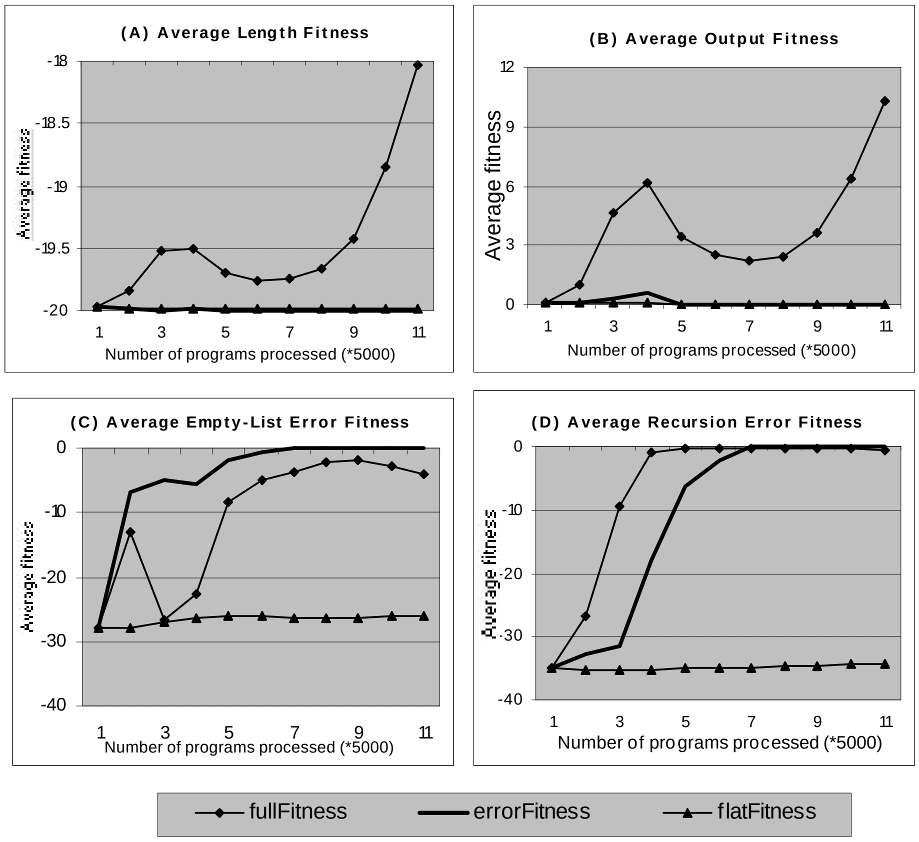

Figure 4 Experimental Results for the MAP program.

After most of the programs with recursion error were eliminated from the population, empty-list error fitness became the dominant factor to drive evolution. This was shown by the fast improvement of empty-list error fitness between the processing of 20,000 and 35,000 programs (see Fig 4(C)) and the decrease of both output and length fitness during this period of time (see Figure 4 (A) & (B)).

After 35,000 programs were processed, the population contained no program with recursion errors. Since the empty-list error fitness was reasonably good (-5), the evolution focused on improving output and length fitness. As shown in Figure 4 (A) & (B), their fitness values increased very fast. As a result, 3 of the 10 runs generated correct `map` programs.

When genetic operations were directed to evolve programs which satisfy the two error constraints (errorFitness), both the average empty-list and recursion fitness improved (see Figure 4 (C) & (D)). However, similar to the results of `nth` experiment, the average length and output fitness did not improve at all throughout the runs. Consequently, no solution was found at the end of the 10 runs. Applying genetic operations on randomly selected programs (flatFitness) did not generate better programs than those created using random search (initial population). Figure 4 (A), (B), (C) & (D) show that the four average fitness values stayed pretty much the same as the initial population throughout the run. Consequently, no solution was found at the end of the 10 runs.

6 Discussion

Based on the experimental results, we will make a determination of what has contributed to the success of the generation of the two polymorphic programs.

Random generation of 100,000 type-correct programs could not find a correct `nth` nor `map` program. Moreover, applying genetic operations on randomly selected programs did not provide better overall fitness than random search (initial generation). This can be explained by perceiving genetic operations as performing random search within the population. Instead of the whole search space of all possible solution to explore, genetic operations are restricted within the population. Such constraint has been shown to provide no advantage over random search for these two problems. Based on the results, we conclude that polymorphism alone (random sampling of the set of type-correct programs) can not generate correct `nth` nor `map` programs.

Genetic operations guided by a fitness function towards programs which satisfy error constraints have successfully eliminated all programs with these two errors from the population. This leads to the random application of genetic operations within the set of type-correct programs that satisfy the two error constraints. Unfortunately, this restricted set has reduced program diversity and prevented the improvement of population fitness. Based on the results, we conclude that `nth` and `map` programs can not be generated by the combination of polymorphism and error constraint handling.

Only when genetic operations are guided by a fitness function towards programs which not only satisfy the two error constraints but also generate correct output is the PolyGP system able to generate correct `nth` and `map` programs. This result indicates that the success of PolyGP in generating these two programs is due to the combination of polymorphism and GP evolutionary search. Polymorphism ensures that only type-correct programs are generated while GP evolutionary search promotes the programs which most satisfy the fitness function to emerge. Consequently, the two polymorphic programs are generated at the end of the GP search process.

7 Conclusions

We have presented the concept of types in GP and introduced a typed GP system, PolyGP, that provides polymorphism through the use of three different kinds of type variables. This implementation differs from others in that 1) polymorphism is for an infinite number of types, 2) polymorphism not only exists at program creation time but also at program evolution time and 3) polymorphism is extended to function type. This system is therefore strictly more powerful than other polymorphic GP systems.

The usefulness of this implementation is demonstrated by evolving two polymorphic programs (`nth` and `map`), which are in a class of programs we believe to be difficult for other polymorphic GP systems to evolve. We have purposely designed and implemented a series of experiments to assure that type-variable-polymorphism is indeed helpful to the GP evolutionary process. The experimental results confirm our hypothesis. PolyGP has enhance the applicability of GP to a new class of problems that are difficult for other polymorphic GP systems to solve.

Acknowledgments

I would like to thank the anonymous reviewers for their insightful comments. Thanks also to Wes Johnston for proof reading this work.

Bibliography

[1] Brave, S.: Evolving recursive programs for tree search. In: *Advances in Genetic Programming II*. P. J. Angeline and K. E. Kinnear, Jr. (eds.), page 203-219. 1996.

[2] Cardelli, L. and Wegner, P.: On understanding types, data abstraction, and polymorphism. In: *Computing Surveys*, Vol. 17:4, pages 471-522. 1985.

[3] Haynes, T. D., Schoenefeld, D. A. and Wainwright, R. L.: Type inheritance in strongly typed genetic programming. In: *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.), pages 359-376. 1996.

[4] Haynes, T. D., Wainwright, R. L., Sandip, S. and Schoenefeld, D.: Strongly typed genetic programming in evolving cooperation strategies. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 271-278, 1995.

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

[6] Langdon, W. B.: Scaling of program fitness spaces. In: *Evolutionary Computation*, Vol. 7 No. 4, pages 399-428, 1999.

[7] Luke, S.: personal communication, 2000.

[8] McPhee, N. F., Hopper, N. J. and Reierson, M. L.: Impact of types on essentially typeless problems in GP. In: *Genetic Programming 1998: Proceedings of the Third Annual Conference.* pages 232-240. 1998.

[9] Montana, D. J.: Strongly typed genetic programming. In: *Evolutionary Computation*, Vol. 3:2, pages 199-230. 1995.

[10] M. L. Wong and K. S. Leung. An adaptive inductive logic programming system using genetic programming. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. pages 737-752, 1995.

[11] Yu, T. and Bentley, P.: Methods to evolve legal phenotypes. In: *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature*. Lecture Notes in Computer Science, Vol. 1498, Springer-Verlag, Berlin, pages 280-291, 1998.

[12] Yu, T. and Clack, C.: PolyGP: a polymorphic genetic programming system in haskell. In: *Proceedings of the Third Annual Genetic Programming Conference*, page 416-421, 1998.

[13] Yu, T.: *An Analysis of the Impact of Functional Programming Techniques on Genetic Programming*, Phd Thesis, University College London, 1999.