Performance Enhanced Genetic Programming

Chris Clack and Tina Yu
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, England
C.Clack@cs.ucl.ac.uk T.Yu@cs.ucl.ac.uk

Abstract

Genetic Programming is increasing in popularity as the basis for a wide range of learning algorithms. However, the technique has to date only been successfully applied to modest tasks because of the performance overheads of evolving a large number of data structures, many of which do not correspond to a valid program. We address this problem directly and demonstrate how the evolutionary process can be achieved with much greater efficiency through the use of a formally-based representation and strong typing. We report initial experimental results which demonstrate that our technique exhibits significantly better performance than previous work.

1 Introduction

Genetic Programming (GP) automatically generates a computer program to solve a specified problem. It does this by using a genetic algorithm to search through a space of possible programs for one that is nearly optimal in its ability to solve the particular problem. Each program is represented by a data structure which gives the syntax structure (the "parse tree") for that program. An initial population is created and is then evolved through a process of genetic crossover and mutation. A "fitness" function acts as an interpreter to evaluate each parse tree and scores each program according to how well it solves the problem. Low-scoring programs are no longer considered as part of the population; either a new population is constructed, or a steady-state population is maintained by discarding enough low-scoring programs to balance the newly created programs. Evolution is an iterative process of crossover, mutation, and fitness testing until a termination criterion is met. The best parse tree is normally chosen for the target application. Thus, there are three main phases in the GP technique: (1) constructing the initial population; (2) testing each program for "fitness" and discarding unfit programs; (3) using genetic crossover and mutation to create new programs. Phases 2 and 3 are iterated until completion.

Despite its recent popularity, the GP technique has to date only been successfully applied to modest tasks. The problems with traditional GP are: (1) the search space is very large due to the large parse trees with large tree depth; (2) many parse trees do not correspond to valid programs (the initial population is created at random, and the genetic crossover and mutation create random programs).

Montana [10] has illustrated how the size of the search space of possible parse trees depends upon the number of terminals (T), the number of functions (F), and the maximum tree depth (D); even for small problems (for example, T=2, F=5, D=6), Montana calculates a search space of roughly 10^{27} parse trees. Furthermore, only about 10^{12} of these correspond to valid (type-correct) programs. Recent GP research has investigated ways to reduce the size of parse trees and to reduce the size of the search space in order to make GP more efficient:

1. **reusable program parts**. This includes research into Automatically Defined Functions [8], Module Acquisition[1,5] and the use of Recursion [2]. Also, Montana [10] argues that generic functions and generic data types can help to reduce program size by providing abstraction of program parts.
2. **type constraints** ensure that only type-correct programs are generated for the initial population and during genetic crossover and mutation. This approach was originally suggested by Koza [7] and then extended by Montana who has developed the idea of Strongly Typed Genetic Programming [10].

We address the problem of GP performance by building on Montana's Strongly Typed Genetic Programming. Our research differs from other work in four important ways:

1. we represent programs using a compact expression-based parse-tree;
2. we use a type unification algorithm rather than table lookup to instantiate type variables;
3. we support reuse through polymorphic types (similar to Montana's generic types) and higher-order functions (which improve on Montana's generic functions);
4. our tree representation permits crossover on partial applications of functions, thereby providing more diversity in the evolutionary process.

We achieve a reduced search space in a similar way to Montana, with similar advantages accruing through the use of polymorphic types, yet we require smaller population size and our parse trees have reduced depth. As a result, our system evolves faster than Montana's. We demonstrate this fact by repeating two of Montana's experiments and providing our initial experimental results.

The paper is structured as follows: Section 2 summarises related work; Section 3 explains our new parse-tree representation; Section 4 discusses our new type system; Section 5 presents our experimental results; Section 6 analyses those results and Section 7 concludes and provides suggestions for future work.

2 Related Work

The idea of constraining the syntactic structure of the parse-tree data structures has been advocated by Koza and then developed further by Montana. In this section we very briefly outline these two important pieces of related work.

2.1 Constrained Syntactic Structure

With Koza's "Constrained Syntactic Structure"[7], a set of rules is defined to specify which terminals and functions are allowed to be the child nodes of every function in the parse tree. When generating a new parse tree and when performing genetic operations, these rules are applied to ensure the syntactic constraints. There are many similarities between Koza's approach and the program structuring requirements of a simple monomorphic type system.

2.2 Strongly Typed Genetic Programming

Montana[10] has developed Koza's original approach so that constraints are given indirectly, through a type system, rather than directly (per-function). Furthermore, Montana has extended this notion to include generic functions and generic data types.

Montana maintains a table giving the types of all available terminals and functions. Thus, if a function takes an argument of type X then this implicitly constrains its child to produce a value of type X. A second table provides type constraints ("type possibilities") according to the depth in the tree where type matching occurs: this extra information constrains the choice of function to create nodes in the tree to ensure that the tree can grow to its maximum depth. During the creation of the initial population (and during crossover and mutation), each parse tree is grown top-down by choosing functions and terminals at random within the constraints of the types in the table. In this way, the initial population only consists of parse trees that are type-correct.

The major contributions of Montana's work are the use of generic functions and generic data types. They provide a form of parametric polymorphism and strong typing: hence "Strongly Typed Genetic Programming".

However, we believe there are two key problems with Montana's work: (1) The technique exhibits poor performance during tree creation due to the required table-lookup for type possibilities; (2) Although Montana is able to grow a version of `MAPCAR`, the function-type argument is treated in an ad-hoc manner and there is no general support for growing functions which take functions as

arguments and/or return functions as results.

3 A New Representation

We present a new technique for Strongly Typed Genetic Programming (STGP) which uses an expression-based (rather than statement-based) parse-tree representation; this leads to smaller tree depth. The evolution performance is improved through smaller search space.

This new technique exhibits higher performance than Montana's STGP, whilst also provides greater reuse (through the support of higher-order functions) and greater diversity (through the support of crossover on partial applications). Furthermore, the use of a formally-based representation (our parse tree syntax is based on the λ-calculus) and a formally-based type system (our type system is loosely based on the Hindley/Milner polymorphic type system [9] and we use Robinson's unification algorithm [11]) will provide greater confidence in the correctness, completeness and expressiveness of our GP system. However, it is not the purpose of this paper to explore the formal basis underlying our work.

3.1 Representation

The parse trees used in traditional GP are often represented as S-Expressions, for example Montana's "`NTH`" function which gets the N-th element of a list: `(EXECUTE-TWO (DO-TIMES (EXECUTE-TWO (SET-VAR-1 L) N) (SET-VAR-1(CDR GET-VAR-1)))(CAR GET-VAR-1).`

The above expression is based on statements with multiple assignment (such as `SET-VAR-1`) and explicit sequencing (such as `EXECUTE-TWO`); we call this a *statement-based* syntax tree. Unfortunately, statement-based syntax trees are not ideal for strongly-typed GP for the following two reasons: (1) Both assignment and sequencing statements require the type system to be extended with a `VOID` type to support nodes which do not return a value. This can slow down tree construction since special rules must be applied to cater for this additional type; (2) The assignment statement causes the search space to be larger than necessary. This is because `GET-VAR` can legally be applied to a variable that has not yet been set: this should be detected as an error, yet it is very difficult for the statement-based type system to detect this error. Thus, the search space contains invalid trees.

In order to enhance the performance of STGP, we use *expression-based* parse trees, where there are no re-assignments (only initial assignments, called bindings), no sequencing other than that implied by the data-dependencies of the operators, and where iteration is achieved through the use of recursion rather than operators such as `DO-TIMES`. The resulting type system is less complex (for example, we no longer need a `VOID` type) and more rigorous (more invalid programs are detected). In this way we aim to achieve a smaller search space and higher performance than Montana.

Our abstract language syntax follows that of the λ-calculus; a program is an expression, defined as:

expression	::	c	c*onstant (such as 0,'A', TRUE)*
		| x	i*dentifier*
		| p	*built-in (primitive) operator*
		| exp1 exp2	*application of one expression to another*

Note that we do not support λ-abstraction; since we wish to compare our work directly with Montana's we chose not to complicate the parse tree with local function definitions. In our experiments we attempt to grow programs with a pre-defined number of arguments, each with a pre-defined identifier.

The first two expressions (`c, x`) are terminals (or "leaves") in a parse tree, whereas the others are functions (or "nodes"). The concrete language syntax and representation is superficially similar to Montana's in that we use S-Expressions, but thereafter the similarity ends. Here is our version of the NTH program: `(IF (= N 0) (CAR L) (NTH (- N 1) (CDR L)))`

Just as neither version is explicit about how the arguments `N` and `L` are bound to their actual arguments, our version is not explicit about the binding of the name `NTH`. Nevertheless, the two

examples compute the same result.

3.2 Partial Applications

In our abstract expression syntax, function application is *curried*: that is, a function is applied to one argument at a time, thus allowing partial application to be expressed. This curried style of function application means that a function, such as `IF`, which takes more than one argument is applied to its arguments one at a time, with an intermediate function implied at each stage.

For example, `(IF (= N 0))` implies an intermediate function which is expecting two more arguments, and `((IF (= N 0)) (CAR L))` implies an intermediate function which is expecting one more argument. Finally, the expression `(((IF (= N 0)) (CAR L)) (NTH (- N 1) (CDR L)))` is fully applied and returns a result. This style of function application provides greater opportunity for crossover (at the partial application points) operations to manipulate genetic material in a more diverse way to create new programs. Thus a smaller population, i.e. less genetic material, is required for the evolution process.

3.3 Recursion

Our treatment of recursion is very simple. We give a name to the program that we are growing; at any point within the parse tree it can use its own name just as if it were a built-in function (this can be seen in action in our early example of the "`NTH`" program).

We limit the number of recursive calls (by the length of the input list) to avoid infinite loops [2]. When the limit is reached, the interpreter aborts with a flag that is used in the computation of the fitness value of the program.

4 A New Type System

Our new technique requires the use of a type system in order to ensure that invalid parse trees are never created. We define a syntax of valid types which annotate the nodes and leaves of the parse tree; these are used during tree creation and during crossover and mutation. We also use a type unification algorithm to determine whether two types are contextually equivalent (which is not straightforward, since we allow type variables in order to achieve polymorphism).

4.1 Type Syntax

Our abstract type syntax is given by:

$\sigma :: \tau$	*built-in type*
$\mid \upsilon$	*type variable*
$\mid \sigma_1 \text{ -> } \sigma_2$	*function type*
$\mid [\sigma_1]$	*list of elements all of type* σ_1
$\mid (\sigma_1 \text{ -> } \sigma_2)$	*bracketed function type*

$\tau ::$ `int` | `string` | `bool` | $\texttt{generic}_i$

$\upsilon ::$ $\texttt{dummy}_i$ | $\texttt{temporary}_i$

Our type system improves upon Montana's Generic Functions by supporting the use of higher-order functions (Montana does not implement function arguments).

Higher-Order Functions

Higher-order functions are those which can take other functions as arguments and/or return functions as results, whereas Generic Functions are function *templates* which may take function arguments (though Montana does not implement this). Generic Functions must have their arguments instantiated whenever the template is used in the parse tree. By contrast, higher-order functions need not have their

arguments instantiated: the choice of function to be passed as an argument can for example be made at evaluation time as the result of a conditional test. Higher-order functions provides better abstraction and reduce program size.

In our type system, the passing of a function as an argument is indicated by the use of the bracketed function type. The brackets can also be used to indicate the use of a function as a return type (though this is not strictly necessary).

Polymorphism

Polymorphic functions are those whose parameters may have more than one type, thereby providing a way to abstract an algorithm for reuse. For example, the build-in function `CAR` takes a list of arbitrary type as argument and returns the first element of the list. Our system provides three kinds of type variable, as follows:

Generic Type Variables: The "generic" types are used when evolving polymorphic programs such as `NTH`, which has type `int -> [G1] -> G1` where `G1` is a type variable. While the program is being evolved the type variable is not instantiated: it therefore takes on the role of a built-in type.

Dummy and Temporary Type Variables: Dummy types express the polymorphism of built-in functions: whenever they are used in a parse tree they must be instantiated to some other type (not a dummy type). Note that if a dummy type variable occurs more than once, then it is necessary to instantiate all occurrences to the same type. Where there are no constraints the dummy type is instantiated as a new temporary type variable. This delayed binding of temporary type variables supports a form of polymorphism within the parse tree as it is being created. A useful invariant is that a dummy type variable will never occur in a parse tree.

4.2 The Initial Environment

It is necessary to supply the system with an initial set of terminals and functions. Their types must also be specified. In Section 4.6, `a` is a dummy type variable, `G1` is a generic type, and the type for `MAPCAR` uses a bracketed function type for the first argument.

4.3 Annotating Expressions with Types

Every expression in the language is annotated with a type:

- Constants such as `0` and identifiers such as `f` have a type pre-defined by the user;
- Built-in functions also have pre-defined types (e.g. `CAR` has the type `[a] -> a`);
- Applications have a type given as follows: if `exp1` has type (σ_1 –> σ_2) and `exp2` has type σ_1 then `(exp1 exp2)` has type σ_2 else there is a type error

4.4 Type Variable Instantiation

When a parse-tree is created, it is grown from the top node downwards. There is a required type for the top node of the tree, and any function can be selected as the top node as long as its return type unifies with the required type. The selected function will require arguments to be created at the next (lower) level in the tree: there will be type requirements for each of those arguments, and again any function can be selected as long as its return type unifies with the required type.

At any point, if we fail to find a function whose return type unifies with the required type, we stop growing the tree by randomly selecting a terminal whose type unifies with the required type. Without any prechecking procedure, this random selection of functions and terminals works well most of the time (85%). To handle the small portion of failing cases, we implement a backtracking mechanism. When the creation of a particular subtree fails, we backtrack to the particular node and regenerate a new subtree. This is an overhead of our type system but, considering the small percentage

failure rate, it's a price worth paying.

When the type for a selected function contains a dummy type variable, then it must be instantiated in a way that satisfies the required type. Furthermore, whenever an occurrence of a dummy type variable is instantiated, all the other occurrences of the same dummy type variable in that type must be instantiated to the same value: this process is called "contextual instantiation" [3] which is performed by a unification algorithm explained in the next section. The following table shows how the dummy type variables can be instantiated:

Table 1:

required type	selected function	instantiated type
`[G2]`	`if`	`bool -> [G2] -> [G2] -> [G2]`
`(G1 -> G2)`	`car`	`[G1 -> G2] -> (G1 -> G2)`
`[bool]`	`cons`	`bool -> [bool] -> [bool]`
`bool`	`=`	`T1 -> T1 -> bool`

Within a parse tree, temporary type variables must be instantiated consistently to maintain the legality of the tree. One tricky situation involves the unification of a dummy type variable and a temporary type variable. In this case, the dummy type variable is first instantiated to a unique temporary type variable before unifying with the other temporary type variable. A global type environment is maintained for each parse tree during the process of tree generation: this environment records how each temporary type variable is instantiated. Once a temporary type variable is instantiated, all occurrences of the same variable are instantiated to the same type.

4.5 Unification Algorithm

Our type system uses Robinson's unification algorithm to instantiate type variables. Briefly, the unification algorithm takes two types and determines whether they unify, i.e. whether they are equivalent in the context of a particular set of instantiations of type variables (called a "substitution", or a "unifier"). If the two type expressions unify, it returns their most general unifier, otherwise it flags an error.

- A substitution, θ, is a finite set (possible empty) of pairs of the form (X_i, t_i) where X_i is a type variable and t_i is a type value or a type variable. For example: θ = { `(a, int)` }.
- The result of applying a substitution θ to a type A, denoted by $A\theta$, is the type obtained by replacing every occurrence of X in A by t, for each pair (X, t) in θ. For example:

  `a-> a {(a, int)}  =  int -> int`
- Two types A and B unify if there exists a substitution θ which makes the types identical $A\theta = B\theta$. For example, if A = `T1-> int` and B = `[string] -> T2`, A and B unify with θ = { `(T1, [string]),(T2, int)` }
- There may be more than one substitution which unifies two types. The "most general unifier" of two types A and B is a substitution θ that unifies A and B such that $A\theta$ is more general than any other common instance of A and B. For example, if $A\theta_1$ = `T1 -> int` and $A\theta_2$ = `int -> int`, then $A\theta_1$ is more general than $A\theta_2$ and θ_1 is the most general unifier.

The unification algorithm is applied in two places in the system: (1) During the instantiation of dummy type variables of a polymorphic function, when dummy type variables get instantiated and bound to type values; (2) During the instantiation of temporary type variables in a parse tree, when temporary

type variables get instantiated and bound to type values.

4.6 An Example

The following demonstrates how the type system works while growing the MAPCAR program:

Assuming the following terminal and function sets, we would like to generate a parse tree with depth level 3 and return type [G2].

```
T = { l:: [G1],   nil:: [a],   f2:: (G1 -> G2)}
F = { f1      :: a   -> a   -> b,
      MAPCAR  :: (G1 -> G2) -> [G1] -> [G2],
      cons    :: a   -> [a] -> [a],
      cdr     :: [a] -> [a]}
```

The return types of all 4 functions unify with the expected type [G2]. The system randomly picks f1. While b is instantiated to [G2], a is instantiated to a temporary type variable T1.

$((f1^{T1->T1->[G2]}\ ARG^{T1})^{T1->[G2]}\ ARG^{T1})^{[G2]}$

We work from right to left to expand the two "ARG" arguments: the next expected type is the second argument type of f1: T1. Again, the return types of all 4 functions unify with T1. We select "cons". Note that the instances of the dummy type variable a are first instantiated to the temporary type variable T2 before unifying with T1. Also, the instantiation of T1 to [T2] is reflected in all the occurrences of T1 throughout the program.

$((f1^{[T2]->[T2]->[G2]}\ ARG^{[T2]})^{[T2]->[G2]}\ ((cons^{T2->[T2]->[T2]}\ ARG^{T2})^{[T2]->[T2]}\ ARG^{[T2]})^{[T2]})^{[G2]}$

The next expected type is the second argument type of cons: [T2]. Since it's leaf level now, we have to select a terminal. Both "l" and "nil" unify with [T2]. We select "l" and unify [T2] with [G1]:

$((f1^{[G1]->[G1]->[G2]}\ ARG^{[G1]})^{[G1]->[G2]}\ ((cons^{G1->[G1]->[G1]}\ ARG^{G1})^{[G1]->[G1]}\ l^{[G1]})^{[G1]})^{[G2]}$

The next expected type is the first argument type of "cons": G1. Notice that it is the type value to which T2 has been instantiated. Again, we are on leaf level and have to select a terminal. Unfortunately there is no terminal whose type unifies with G1. We must therefore back-track one level up and reselect a new function to replace "cons". This time we select "tail", instantiate a to be T2, and unify T1 with [T2]:

$((f1^{[T2]->[T2]->[G2]}\ ARG^{[T2]})^{[T2]->[G2]}\ (tail^{[T2]->[T2]}\ ARG^{[T2]})^{[T2]})^{[G2]}$

The next expected type is the only argument type of "tail": [T2]. We are on leaf level and there are two terminals we can select: "l" and "nil". We select "nil" and instantiate a with T3 before unifying [T3] with [T2]:

$((f1^{[T3]->[T3]->[G2]}\ ARG^{[T3]})^{[T3]->[G2]}\ (tail^{[T3]->[T3]}\ nil^{[T3]})^{[T3]})^{[G2]}$

The next expected type is the first argument type of "f1": [T3]. All 4 functions are legal and we select "MAPCAR". We must therefore unify [T3] with [G2]:

$((f1^{[G2]->[G2]->[G2]}\ ((MAPCAR^{(G1->G2)->[G1]->[G2]}\ ARG^{G1->G2})^{[G1]->[G2]}\ ARG^{[G1]})^{[G2]})^{[G2]->[G2]}\ (tail^{[G2]->[G2]}\ nil^{[G2]})^{[G2]})^{[G2]}$

The next expected type is the second argument type of MAPCAR: [G1]. Terminal "l" is selected:

$((f1^{[G2]->[G2]->[G2]}\ ((MAPCAR^{(G1->G2)->[G1]->[G2]}\ ARG^{G1->G2})^{[G1]->[G2]}\ l^{[G1]})^{[G2]})^{[G2]->[G2]}\ (tail^{[G2]->[G2]}\ nil^{[G2]})^{[G2]})^{[G2]}$

The next expected type is the first argument type of MAPCAR: (G1->G2). "f2" is the only choice and the program is complete:

$((f1^{[G2]->[G2]->[G2]}\ ((MAPCAR^{(G1->G2)->[G1]->[G2]}\ f2^{G1->G2})^{[G1]->[G2]}\ l^{[G1]})^{[G2]})^{[G2]->[G2]}\ (tail^{[G2]->[G2]}\ nil^{[G2]})^{[G2]})^{[G2]}$

4.7 Crossover Example

Once the initial population of expression-based parse trees has been constructed, the iterative process of evolution begins. During evolution, new parse trees are created through *crossover* and *mutation*. When a new tree is created, it's important to discard another tree to conserve memory. We choose to discard those trees with the lowest fitness score.

We restrict crossover to be performed in application nodes only (these include full application and partial application nodes): this is to promote the recombining of large structures[6]. By reference to our abstract syntax, this means that any occurrence of the application expression `(exp1 exp2)` appearing at any place in one tree can be swapped with any application expression in another parse tree.

When performing crossover, we first select a crossover node from one parent. The return type value with the depth of the node is passed over to the second parent to select another crossover node. In the second parent tree, a crossover node will be selected according to two criteria: (1) Its return type value must unify with the given return type value; (2) Its depth must be such that the new tree will satisfy the maximum tree depth. For example, we use the parse tree generated in the previous section as the first parent:

((**f1** `[G2]->[G2]->[G2]` ((**MAPCAR** `(G1->G2)->[G1]->[G2]` **f2** `G1->G2`) `[G1]->[G2]`
l `[G1]`) `[G2]`)**[G2]->[G2]** (tail `[G2]->[G2]` nil `[G2]`) `[G2]`) `[G2]`

The second parent tree is given below. Note that the double-underlined type following the bolded expression in both parents indicates the partial application node where the crossover operation occurs:

(tail `[G2]->[G2]` ((**f1** `[T1]->[T1]->[G2]` **nil** `[T1]`)**[T1]->[G2]** nil `[T1]`) `[G2]`)`[G2]`

The following parse-tree is generated by the crossover operation. Note that `T1` is instantiated to `G2`:

((f1 `[G2]->[G2]->[G2]` nil`[G2]`) `[G2]->[G2]` (tail `[G2]->[G2]` nil `[G2]`) `[G2]`)`[G2]`

5 Experiments

In this section we present our experimental results. First we describe our genetic algorithm and then we present the results of two of Montana's experiments: `NTH-3` and `MAPCAR`.

5.1 The Genetic Algorithm

We follow Montana's STGP in using steady-state replacement[12] to perform population updates. Initially, we create a population with a specified size. Within the population, every tree is unique. During evolution, we select two trees to perform crossover. The system ensures that the newly created tree is unique before putting it back to the same population pool to replace the tree with the lowest fitness score. The size of the population therefore remains constant. The advantage of steady-state replacement is that a tree with a good fitness score is immediately available as a parent for reproduction rather than having to wait until the next generation.

To facilitate direct comparison, we follow Montana's STGP in using exponential fitness normalization [4] to select parents for reproduction. This means: (1) We use rank selection instead of fitness-proportionate selection, and (2) The probability of selecting the *n*-th best individual is `Parent-Scalar` times the probability of selecting the *(n-1)*-th best individual. However, we must adjust `Parent-Scalar` because we use a different population size.

5.2 Handling of Run-Time Errors

When the fitness function evaluates each parse tree, two possible run-time errors can occur. Both run-time errors are reflected in the fitness evaluation function, as will be illustrated below. (1) Non-terminating recursion causes program evaluation to abort with an error flag; (2) Taking the `CAR` or `CDR` of an empty list is handled by using (the evaluation continues). We do this because we believe that even

trees with this type of error may still contain good genetic material. The only way to reuse these good genetic building block is to complete the evaluation and score the program accordingly. The following table gives the default values that we use for handling the empty-list error:

Table 2:

Type	Default	Type	Default	Type	Default
`int`	`0`	`bool`	`False`	`T1`	`0`
`string`	`" "`	`[T1]`	`[]`	`G1`	`0`

If any temporary type variables remain in the tree, it is clear that their value will never be inspected and so we can return a value of any type (we choose int). A similar argument can be used to support the use of 0 for generic type variables.

5.3 The Function NTH-3

Problem Description: The `NTH-3` function takes two arguments, an integer N and a list L, and returns the Nth element of L. If N < 1, it returns the first element of L. If N > length(L), it returns the last element of L.
Output Type: The output has type `G1`
Arguments: The argument `N` has type `int` and the argument `L` has type `[G1]`
Terminal Set: `T = {L:: [G1], N:: int, 1:: int}`
Function Set:
```
F = {HEAD:: [a]->a,IF-THEN-ELSE:: bool->a->a->a,
     TAIL:: [a]->[a],LENGTH:: [a]->int,
     <= :: int->int->bool, > :: int->int->bool,
     - :: int->int->int,NTH-3:: int->[G1]->G1
```
Fitness Function: We used 12 test cases to evaluate performance. Each test case gave N a different value in the range from 0 to 11. For all the test cases, we took L to be a list of length 10 with all of its entries unique. For each test case, we evaluated the tree and compared the return value with the expected result to compute a score. The score for each test case then summed into a total score which becomes the program's fitness score. Our fitness function is the same as Montana's except we penalise for any error.

$$S_L = 10 \cdot 2^{-d} - (10 \cdot \text{rtError}) - (10 \cdot \text{reError})$$

where d is the distance between the correct position and the return value position, rtError is 1 if there is run-time empty-list error, 0 otherwise. reError is 1 if there is non-terminating recursion error, 0 otherwise.
Genetic Parameters: We use a population size of 3,000, a parent scalar of 0.9965, a maximum tree depth of 5 and 100% crossover rate.
Results: We made 10 runs and all of them found an optimal solution. The average number of individuals evaluated before finding an optimal solution was 6,104. The maximum number of evaluations was 9,008 while the minimum number of evaluations was 4,526. Compare with Montana's STGP which has average number of evaluations of 35,280, our approach performs 6 times better. The shortest program we generated to solve the `nth-3` problem has tree-depth 5, terminal set size 3 and function set size 8: `(IF (<= N 1)(HEAD L)(IF (> N (LENGTH L))(NTH-3 (- N 1) L) (NTH-3 (- N 1) (TAIL L))))`

By comparison, Montana's STGP generated a shortest program which has tree-depth 7, terminal set size 3 and function set size 10: `(EXECUTE-TWO (DOTIMES (EXECUTE-TWO(SET-VAR-1`

```
L) (- (MIN N (LENGTH L)) 1)) (SET-VAR-1 (CDR GET-VAR-1)))(CAR GET-
VAR-1))
```

5.4 The Function MAPCAR

Problem Description: The `MAPCAR` function takes two arguments, a function `F` and a list `L`, and returns the list obtained by applying `F` to each element of `L`.
Output Type: The output has type `[G2]`
Arguments: The argument `F` has type `G1->G2` and the argument `L` has type `[G1]`
Terminal Set: `T = {L:: [G1], NIL:: [a], F:: (G1->G2)}`
Function Set: `F = {HEAD ::[a]->a, IF-THEN-ELSE ::bool->a->a->a, TAIL ::[a]->[a], CONS ::a->[a]->[a], NULL ::[a]->bool, F ::G1->G2, MAPCAR ::(G1->G2)->[G1]->[G2]}`

Fitness Function: We used 2 different lists for the argument L and one function for the argument F. The 2 lists were: 1) the empty list and 2) a list with 10 elements whose values were the characters A to J. The F is a function which converts alphabetic characters into numbers, i.e. A to 1, B to 2, C to 3 and so on. For each test case, we evaluated the tree and compared the return list with the expected list. The fitness score was computed based on how close the return list was to the expected list. The score for each test case are summed into a total score which becomes the program's fitness score. The following is our fitness function, which is the same as Montana's except we penalise for any error.

$$S_L = -2 \cdot \left|\text{length}(L) - \text{length}(L_r)\right| + \sum_{e \in L} 10 \cdot (2^{-\text{dist}(e, L_r)})$$

$$-(10 + 2 \cdot \text{length}(L)) \cdot \text{rtError} \; -(10 + 2 \cdot \text{length}(L)) \cdot \text{reError}$$

where $\text{dist}(e,L_r)$ is ∞ if $e \notin L_r$ and otherwise is the distance of *e* from the *e*th position in L_r. The rtError value is 1 if there is a run-time empty-list error, 0 otherwise. The reError value is 1 if there is non-terminating recursion error, 0 otherwise.
Genetic Parameters: We use a population size of 5,000, a parent scalar of 0.999, a maximum tree depth of 5 and 100% crossover rate.
Results: We made 7 runs and all of them found an optimal solution. The average number of individuals evaluated before finding an optimal solution was 28,331. The maximum number of evaluations was 40,601 while the minimum number of evaluations was 14,099. Compare with Montana's STGP which has average number of evaluations of 204,000, our approach performs about 7 times better.The shortest program we generated to solve the `MAPCAR` problem has tree-depth 5, terminal set size 3 and function set size 7: `IF (NULL L) NIL (CONS (F (HEAD L)) (MAPCAR F (TAIL L)))`

By comparison, Montana's STGP generated a shortest program which has tree-depth 8, terminal set size 3 and function set size 10: `(EXECUTE-TWO(DOTIMES(EXECUTE-TWO(SET-VAR-1 L)(LENGTH L))(EXECUTE-TWO(SET-VAR-2(APPEND GET-VAR-2 (FUNARG CAR (GET-VAR-1)))))(SET-VAR-1 (CDR GET-VAR-1)))) (GET-VAR-2))`

6 Analysis

According to our experiments, our technique evolves 6 to 7 times faster than Montana's for two reasons: (1) the search space is smaller due to smaller tree depth value; (2) our initial population size is smaller due to the benefit of partial application crossover. The operation manipulates genetic material in more diverse way to generate new program. The number of genetic material required for evolution process is therefore reduced.

7 Conclusion and Future Work

We have presented a new technique for Strongly Typed Genetic Programming that evolves much faster than previous systems. This work contributes to GP in the following areas:

- We use an expression-based (rather than statement-based) parse-tree representation, which leads to more compact parse trees and smaller search space;
- We use a type system which supports both polymorphic and higher-order types, thereby providing greater reuse than Montana within a single framework;
- We use a type unification algorithm rather than table lookup to instantiate type variables;
- Our tree representation permits crossover on partial applications of functions, thereby requires smaller population size by providing more diversity in the evolutionary process.

We seek to extend our work in the following directions: (1) To conduct more experiments to gain more knowledge about how our type system guides GP in solving problems; (2) To utilize generated polymorphic functions as an intermediate building block for GP to learn other programs (this will provide Automatically Defined Polymorphic Functions).

Acknowledgements

The authors like to thank W.B. Langdon, T. Westerdale and anonymous reviewers for their ideas and criticisms.

References

1. P.J. Angeline. Genetic Programming and Emergent Intelligence. *Advances in Genetic Programming*, K.E. Kinnear, Jr. (ed.), MIT Press, Cambridge, MA, pp. 75-98, 1994.
2. S. Brave. Evolving Recursive Programs for Tree Search. *Advances in Genetic Programming II*, P.J. Angeline and K.E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pp. 203-220, 1996.
3. L. Cardelli. Basic Polymorphic Typechecking. Science of Computer Programming. Vol. 8, pp. 147-172, 1987.
4. A.L. Cox, Jr., L. Davis, & Y. Qiu. Dynamic Anticipatory Routing in Circuit-Switched Telecommunications Networks. *Handbook of Genetic Algorithms*. L. Davis (ed.), Van Nostrand Reinhold, New York, pp.124-143, 1991.
5. K.E. Kinnear, Jr. Alternatives in Automatic Function Definition: A Comparison of Performance. *Advances in Genetic Programming*. K.E. Kinnear, Jr.(ed.), MIT Press, Cambridge, MA, pp. 119-141, 1994.
6. J.R. Koza. Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. Proceedings of the 11th International Conference on Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, Vol. I, pp 768-774, 1989.
7. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.
8. J. R. Koza. *Genetic Programming II*, MIT Press, Cambridge, MA, 1994.
9. R. Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, Vol. 17, pp. 348-375, 1978.
10. D.J. Montana. Strongly Typed Genetic Programming. Journal of Evolutionary Computation, Vol. 3:3, pp. 199-230. 1995.
11. J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. Journal of ACM. Vol. 12:1, pp. 23-49, January 1965.
12. G. Syswerda. Uniform Crossover in Genetic Algorithms. Proceedings of the Third International Conference on Genetic Algorithms and Their Applications, J.D. Schaffer (ed.), Morgan Kaufmann, San Mateo, CA, pp. 2-9, 1989.