BBN Technical Report #7866: Strongly Typed Genetic Programming

David J. Montana
Bolt Beranek and Newman, Inc.
10 Moulton Street
Cambridge, MA 02138

March 25, 1994

Abstract

Genetic programming is a powerful method for automatically generating computer programs via the process of natural selection [Koza 92]. However, it has the limitation known as "closure", i.e. that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. To correct this deficiency, we introduce a variation of genetic programming called "strongly typed" genetic programming (STGP). In STGP, variables, constants, arguments, and returned values can be of any data type with the provision that the data type for each such value be specified beforehand. This allows the initialization process and the genetic operators to only generate syntactically correct parse trees. Key concepts for STGP are generic functions, which are not true strongly typed functions but rather templates for classes of such functions, and generic data types, which are analogous. To illustrate STGP, we present four examples involving vector/matrix manipulation and list manipulation: (1) the multi-dimensional least-squares regression problem, (2) the multi-dimensional Kalman filter, (3) the list manipulation function NTH, and (4) the list manipulation function MAPCAR.

1 Introduction

Genetic programming is a method of automatically generating computer programs to perform specified tasks [Koza 92]. It uses a genetic algorithm to search through a space of possible computer programs for one which is nearly optimal in its ability to perform a particular task. While it was not the first method of automatic programming using genetic algorithms (one earlier approach is detailed in [Cramer 85]), it is so far the most successful. In Section 1.1 we give a very brief overview of genetic algorithms (the unindoctrinated reader is referred to [Goldberg 89] as an introduction). In Section 1.2 we discuss genetic programming and how it differs from a standard genetic algorithm.

1.1 Genetic Algorithms

Genetic algorithms are a class of algorithms for optimization and learning based on the principles of natural evolution. They have been shown to be capable of finding nearly global optima in large and complex spaces

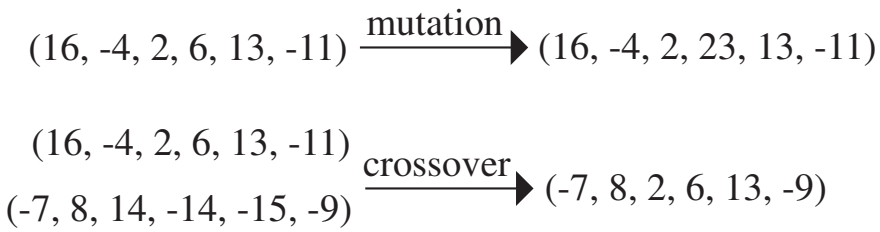

Figure 1: Mutation and crossover for string-based genetic algorithms.

in a relatively short time. Acccording to [Davis 87], a genetic algorithm has five basic components:

1. A representation scheme provides a way to code possible solutions to a problem in a form that is readily manipulable by the genetic operators. The traditional, and still most common, representation is as a fixed-length binary string, but any representation is acceptable as long as there are appropriate genetic operators defined.
2. An evaluation function (or fitness function) assigns a numerical score to any element of the search space. For function optimization problems, the evaluation function is the function that is being optimized.
3. An initialization procedure provides a way to randomly select the individuals (i.e., search points) which constitute the initial population. For a fixed-length string representation, the usual approach is to select each field of each string randomly from its possible values.
4. A set of genetic operators provides a way to use the information from one or more search points to stochastically generate new search points. The two standard genetic operators are mutation and crossover. Mutation takes a single "parent" and produces a "child" which has the same information as the parent in all but a small number of locations, where new information is randomly generated. Crossover takes two parents and produces one or two children whose information is some combination of the information from the parents. Figure 1 shows examples of mutation and crossover acting on fixed-length real-valued strings.
5. A variety of parameter settings determine the run-time characteristics of the genetic algorithm. Such parameters include POPULATION-SIZE, GENERATION-SIZE (when the generation size equals the population size there is full generational replacement and when the generation size is one it is a steady-state genetic algorithm), the operator selection probabilities, and parameters involved with parent selection.

Given these five components, a genetic algorithm operates according to the following steps (and as pictured in Figure 2):

1. Using the initialization process, generate an initial population of size POPULATION-SIZE, and evaluate these individuals.
2. Generate a new generation of size GENERATION-SIZE via the process of reproduction. Creation of a single new inidividual occurs as follows:

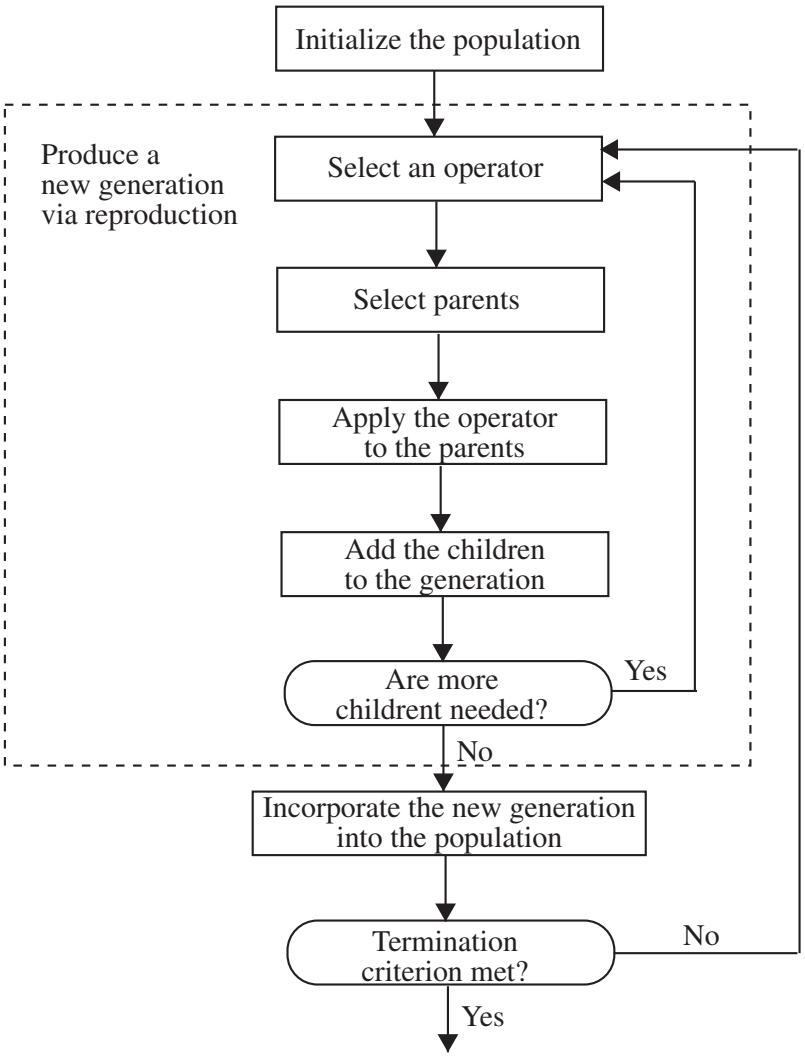

Figure 2: Control flow for a genetic algorithm.

(a) Randomly select an operator according to the operator selection probabilities.

(b) Randomly select parents from the current population with the probability of selecting a particular individual monotonically increasing with the fitness of the individual. (The number of parents to be selected is determined by the operator.)

(c) Apply the operator to the parents to create one or more children.

3. Remove the GENERATION-SIZE worst member of the current population and replace them with the new generation.

4. If a termination criterion is not met, repeat from step (2).

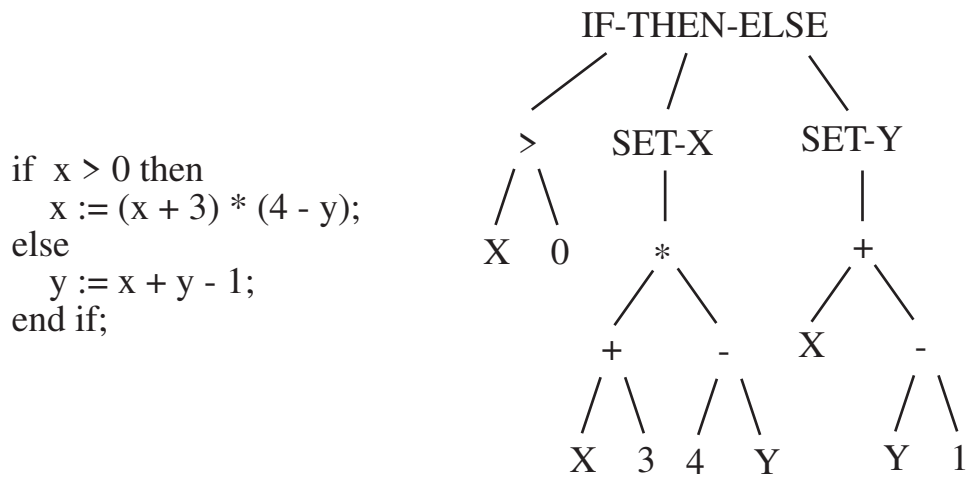

Figure 3: A subroutine and an equivalent parse tree.

1.2 Genetic Programming

We now describe the five components of the type of genetic algorithm used for genetic programming:

(1) **Representation -** For genetic programming, computer programs are represented as parse trees. A parse tree is a tree whose nodes are procedures, functions, variables and constants. The subtrees of a node in a parse tree represent the arguments to the procedure or function of that node. (Since variables and constants take no arguments, their nodes can never have subtrees, i.e. they always are leaves.) Executing a parse tree means executing the root of the tree, which executes its children nodes as appropriate, and so on recursively.

Any subroutine can be represented as a parse tree. For example, the subroutine shown in Figure 3 is represented by the parse tree shown in Figure 3. While the conversion from a subroutine to its parse tree is non-trivial in languages such as C, Pascal and Ada, in the language Lisp a subroutine (which in Lisp is also called an S-expression) essentially is its parse tree, or more precisely is its parse tree expressed in a linear fashion. Each node representing a variable or a constant is expressed as the name of the variable or value of the constant. Each node representing a function is expressed by a '(' followed by the function name followed by the expressions for each subtree in order followed by a ')'. A Lisp S-expression for the parse tree of Figure 3 is

```
(IF-THEN-ELSE (> X 0) (SET-X (* (+ X 3) (- 4 Y))) (SET-Y (+ X (- Y 1))))
```

Because S-expressions are a compact way of expressing subtrees, parse trees are often written using S-expressions (as we do in Section 3), and we can think of the parse trees learned by genetic programming as being Lisp S-expressions.

For genetic programming, the user defines all the possible functions, variables and constants that can be used as nodes in a parse tree. Variables, constants, and functions which take no arguments are the leaves of the possible parse trees and hence are called "terminals". Functions which do take arguments, and

therefore are the branches of the possible parse trees, are called "non-terminals". The set of all terminals is called the "terminal set", and the set of all non-terminals is called the "non-terminal set".

[An aside on terminology: We use the term "non-terminal" to describe what Koza [92] calls a "function". This is because a terminal can be what standard computer science nomenclature would call a "function", i.e. a subroutine that returns a value.]

An important constraint on the user-defined terminals and non-terminals is called **closure**. Closure means that all these elements take arguments of a single data type (e.g., a scalar) and return values of this same data type. This implies that all elements return values that can be used as arguments for any element; hence, any element can be a child node in a parse tree for any other element without having conflicting data types. Koza [92] describes a way to relax this constraint of closure with the concept of "constrained syntactic structures". He uses tree generation routines which only generate legal trees and uses operators which maintain legal syntactic structure. STGP, which is the focus of this paper, builds on and generalizes this concept.

The search space is the set of all parse trees which use only elements of the non-terminal set and terminal set and which are legal (i.e., have the right number of arguments for each function) and which are less than some maximum depth. This limit on the maximum depth is a parameter which keeps the search space finite and prevents trees from growing to an excessively large size.

(2) **Evaluation Function -** The evaluation function consists of executing the program defined by the parse tree and scoring how well the results of this execution match the desired results. The user must supply the function which assigns a numerical score to how well a set of derived results matches the desired results.

(3) **Initialization Procedure -** Koza [92] defines two different ways of generating a member of the initial population, the "full" method and the "grow" method. For a parse tree generated by the full method, the length along any path from the root to a leaf is the same nomatter which path is taken, i.e. the tree is of full depth along any path. Parse trees generated by the grow method need not satisfy this constraint. For both methods, each tree is generated recursively using the following algorithm described in pseudo-code:

```
Generate_Tree( max_depth, generation_method )
begin
    if max_depth = 1 then
        set the root of the tree to a randomly selected terminal;
    else if generation_method = full then
        set the root of the tree to a randomly selected non-terminal;
    else
        set the root to a randomly selected element which is either
            terminal or non-terminal;
    for each argument of the root, generate a subtree with the call
        Generate_Tree( max_depth - 1, generation_method );
end;
```

The standard approach of Koza to generating an initial population is called "ramped-half-and-half". It

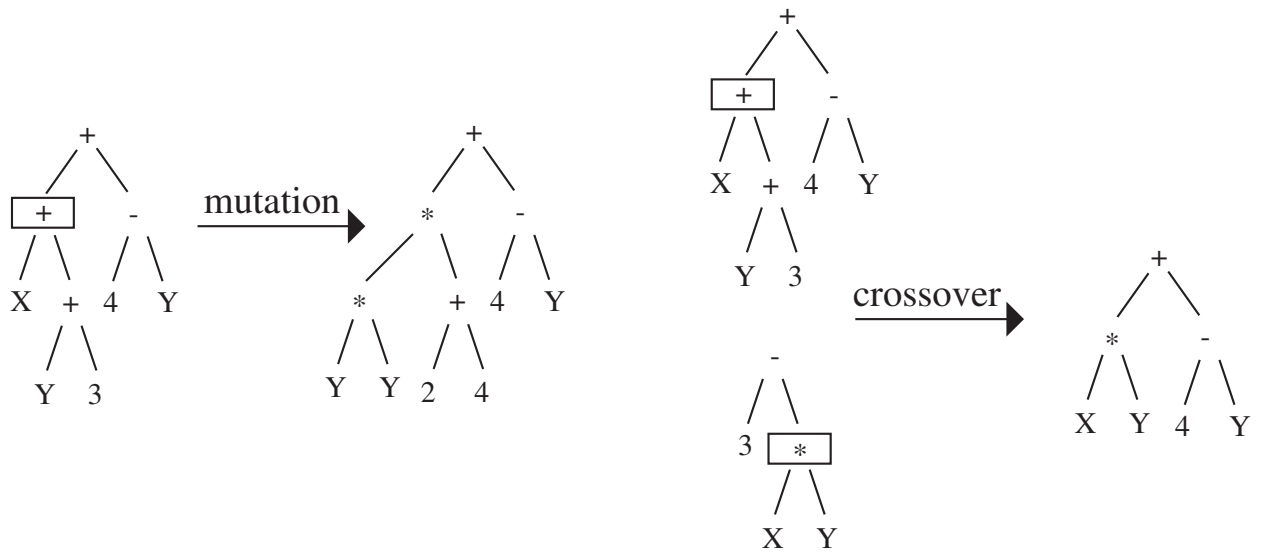

Figure 4: Mutation and crossover for genetic programming.

uses the full method to generate half the members and the grow method to generate the other half. The maximum depth is varied between two and MAX-INITIAL-TREE-DEPTH. This approach generates trees of all different shapes and sizes.

(4) **Genetic Operators -** Like a standard genetic algorithm, the two main genetic operators are mutation and crossover (although Koza [92] claims that mutation is generally unnecessary). However, because of the tree-based representation, these operators must work differently from the standard mutation and crossover. Mutation works as follows: (i) randomly select a node within the parent tree as the mutation point, (ii) generate a new tree of maximum depth MAX-MUTATION-TREE-DEPTH, (iii) replace the subtree rooted at the selected node with the generated tree, and (iv) if the maximum depth of the child is less than or equal to MAX-TREE-DEPTH, then use it. (If the maximum depth is greater than MAX-TREE-DEPTH, then one can either use the parent (as Koza does) or start again from scratch (as we do).) The mutation process is illustrated in Figure 4.

Crossover works as follows: (i) randomly select a node within each tree as crossover points, (ii) take the subtree rooted at the selected node in the second parent and use it to replace the subtree rooted at the selected node in the first parent to generate a child (and optionally do the reverse to obtain a second child), and (iii) use the child if its maximum depth is less than or equal to MAX-TREE-DEPTH. The crossover procedure is illustrated in Figure 4.

(5) **Parameters -** There are some parameters associated with genetic programming beyond those used with standard genetic algorithms. MAX-TREE-DEPTH is the maximum depth of any tree. MAX-INITIAL-TREE-DEPTH is the maximum depth of a tree which is part of the initial population. MAX-MUTATION-TREE-DEPTH is the maximum depth of a subtree which is generated by the mutation operator as the part of the child tree not in the parent tree.

2 Strongly Typed Genetic Programming (STGP)

We now discuss an extension of the basic genetic programming approach, called strongly typed genetic programming (STGP), which is a generalization of Koza's constrained syntactic structures. The key contribution of STGP is that it eliminates the closure constraint described above and hence allows functions which take arguments of any data type and return values of any data type. In its simplest form, STGP does this by requiring that each function specify precisely the data types of its arguments and its returned values (i.e., that the functions be "strongly typed"). STGP can then ensure that all the parse trees it generates satisfy the constraint that the arguments to all functions are of the correct type. With the concept of generics, STGP allows significant relaxation of the need to specify data types precisely; instead, these data types can be of a broad class and specified precisely by the context.

Section 2.1 discusses the details of the extensions from basic genetic programming needed to ensure that all the argument types are correct. Section 2.2 describes a key concept for making STGP easier to use, generic functions, which are not true strongly typed functions but rather templates for classes of strongly typed functions. Section 2.3 discusses generic data types, which are classes of data types. Section 2.4 examines a special data type, called the "VOID" data type, which indicates that no data is returned. Section 2.5 describes the concept of local variables in STGP. Section 2.6 tells how STGP handles errors. Finally, Section 2.7 discusses how our work on STGP has started laying the foundations for a new computer language which is particularly suited for automatic programming.

Note that a different approach to extending genetic programming to allow for different data types is the stack-based approach described in [Perkis 93]. In this approach, different data types are stored on different stacks and manipulated differently based on their being on different stacks as opposed to STGP's method of reasoning about legality based on data types.

2.1 The Basics

We now discuss in detail the changes from standard genetic programming for each genetic algorithm component:

(1) **Representation -** In STGP, unlike in standard genetic programming, each variable and constant has an assigned type. For example, the constants 2.1 and π have the type FLOAT, the variable $V1$ might have the type VECTOR-3 (indicating a three-dimensional vector), and the variable $M2$ might have the type MATRIX-2-2 (indicating a 2x2 matrix).

Furthermore, each function has a specified type for each argument and for the value it returns. Figure 5 shows a variety of strongly typed functions with their argument types and returns types. For those readers not familiar with Lisp, CAR is a function which takes a list and returns the first element [Steele 84]. In STGP, unlike in Lisp, a list must contain elements all of the same type so that the return type of CAR (and other functions returning an element of the list) can be deduced. (Note that below we will describe generic functions, which provide a way to define a single function instead of many functions which do essentially the same operation, e.g.DOT-PRODUCT instead of DOT-PRODUCT-i for i different values.)

To handle multiple data types, the definition of what constitutes a legal parse tree has a few additional

Function Name	Arguments	Return Type
DOT-PRODUCT-3	VECTOR-3 VECTOR-3	FLOAT
VECTOR-ADD-2	VECTOR-2 VECTOR-2	VECTOR-2
MAT-VEC-MULT-4-3	MATRIX-4-3 VECTOR-3	VECTOR-4
CAR-FLOAT	LIST-OF-FLOAT	FLOAT
LENGTH-VECTOR-4	LIST-OF-VECTOR-4	INTEGER
IF-THEN-ELSE-INT	BOOLEAN INTEGER INTEGER	INTEGER

Figure 5: Some strongly typed functions with their argument types and return types.

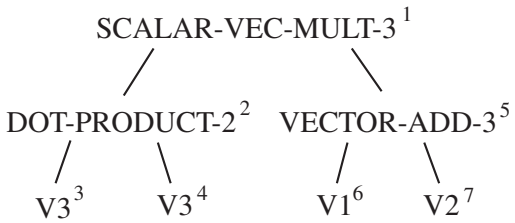

Figure 6: An example of a legal tree for return type VECTOR-3.

criteria beyond those required for standard genetic programming, which are: (i) the root node of the tree returns a value of the type required by the problem, and (ii) each non-root node returns a value of the type required by the parent node as an argument. These criteria for legal parse trees are illustrated by the following example:

Example 1 Consider a non-terminal set $\mathcal{N}$ = {DOT-PRODUCT-2, DOT-PRODUCT-3, VECTOR-ADD-2, VECTOR-ADD-3, SCALAR-VEC-MULT-2, SCALAR-VEC-MULT-3} and a terminal set $\mathcal{T}$ = {V1, V2, V3}, where V1 and V2 are variables of type VECTOR-3 and V3 is a variable of type VECTOR-2. Let the required return type be VECTOR-3. Then, Figure 6 shows an example of a legal tree. Figure 7 shows two examples of illegal trees, the left tree because its root returns the wrong type and the right tree because in three places the argument types do not match the return types.

(2) **Evaluation Function -** There are no changes to the evaluation function.

(3) **Initialization Procedure -** The one change to the initialization procedure is that, unlike in standard genetic programming, there are type-based restrictions on which element can be chosen at each node. One

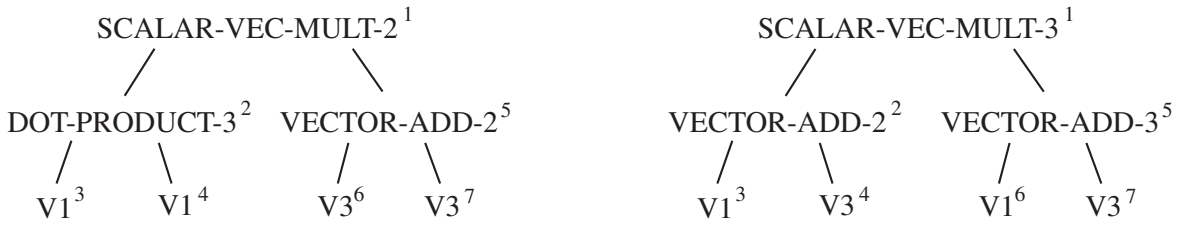

Figure 7: Two examples of illegal trees for return type VECTOR-3.

restriction is that the element chosen at that node must return the expected type (which for the root node is the expected return type for the tree and for any other node is the argument type for the parent node). A second restriction is that, when recursively selecting nodes, we cannot select an element which makes it impossible to select legal subtrees. (Note that if it is impossible to select any tree for the specified depth and generation method, then no tree is returned, and the initialization procedure proceeds to the next depth and generation method.) We discuss this second restriction in greater detail below but first give an example of this tree generation process.

Example 2 Consider using the full method to generate a tree of depth 3 returning type VECTOR-3 using the terminal and non-terminal sets of Example 1. We now give a detailed description of the decision process that would generate the tree in Figure 6. At point 1, it can choose either SCALAR-VEC-MULT-3 or VECTOR-ADD-3, and it chooses SCALAR-VEC-MULT-3. At point 2, it can choose either DOT-PRODUCT-2 or DOT-PRODUCT-3 and chooses DOT-PRODUCT-2. At points 3 and 4, it can only choose V3, and it does. At point 5, it can only choose VECTOR-ADD-3. (Note that there is no tree of depth 2 with SCALAR-VEC-MULT-3 at its root, and hence SCALAR-VEC-MULT-3 is not a legal choice even though it returns the right type.) At points 6 and 7, it can choose either V1 or V2 and chooses V1 for point 6 and V2 for point 7.

Regarding the second restriction, we observe that a non-terminal element can be the root of a tree of maximum depth i if and only if all of its argument types can be generated by trees of maximum depth $i-1$. To check this condition efficiently, we use "types possibilities tables", which we generate before generating the first tree. Such a table tells for each i = 1,...,MAX-INITIAL-TREE-DEPTH what are the possible return types for a tree of maximum depth i. There will be two different types possibilities tables, one for trees generated by the full method and one for the grow method. Example 4 below shows that these two tables are not necessarily the same. The following is the algorithm in pseudo-code for generating these tables.

```
-- the trees of depth 1 must be a single terminal element
loop for all elements of the terminal set
    if table_entry( 1 ) does not yet contain this element's type
        then add this element's type to table_entry( 1 );
end loop;

loop for i = 2 to MAX_INITIAL_TREE_DEPTH
```

```
    -- for the grow method trees of size i-1 are also valid trees of size i
    if using the grow method
        then add all the types from table_entry( i-1 ) to table_entry( i );
    loop for all elements of the non-terminal set
        if this element's argument types are all in table_entry( i-1 ) and
           table_entry( i ) does not contain this element's return type
            then add this element's return type to table_entry( i );
    end loop;
end loop;
```

Example 3 For the terminal and non-terminal sets of Example 1, the types possibilities tables for both the full and grow method are

```
table_entry( 1 ) = { VECTOR-2, VECTOR-3 }
table_entry( i ) = { VECTOR-2, VECTOR-3, FLOAT } for i > 1
```

Note that in Example 1, when choosing the node at point 5, we would have known that SCALAR-VEC-MULT-3 was illegal by seeing that FLOAT was not in the table entry for depth 1.

Example 4 Consider the case when $\mathcal{N}$ = {MAT-VEC-MULT-3-2, MAT-VEC-MULT-2-3, MATRIX-ADD-2-3, MATRIX-ADD-3-2} and $\mathcal{T}$ = {M1, M2, V1}, where M1 is of type MATRIX-2-3, M2 is of type MATRIX-3-2, and V1 is of type VECTOR-3. Then, the types possibilities tables for the grow method is

```
table_entry( 1 ) = { VECTOR-3, MATRIX-3-2, MATRIX-2-3 }
table_entry( i ) = { VECTOR-2, VECTOR-3, MATRIX-3-2, MATRIX-2-3 } for i > 1
```

and the types possiblities table for the full method is

```
table_entry( i ) = { VECTOR-3, MATRIX-3-2, MATRIX-2-3 } for i odd
table_entry( i ) = { VECTOR-2, MATRIX-3-2, MATRIX-2-3 } for i even
```

(4) **Genetic Operators -** The genetic operators, like the initial tree generator, must respect the enhanced legality constraints on the parse trees. Mutation uses the same algorithm employed by the initial tree generator to create a new subtree which returns the same type as the deleted subtree and which has internal consistency between argument types and return types (see Figure 8). If it is impossible to generate such a tree, then the mutation operator returns either the parent or nothing.

Crossover now works as follows. The crossover point in the first parent is still selected randomly from all the nodes in the tree. However, the crossover point in the second parent must be selected so that the subtree returns the same type as the subtree from the first parent. Hence, the crossover point is selected randomly from all nodes satisfying this constraint (see Figure 9). If there is no such node, then the crossover operator returns either the parents or nothing.

(5) **Parameters -** There are no changes to the parameters.

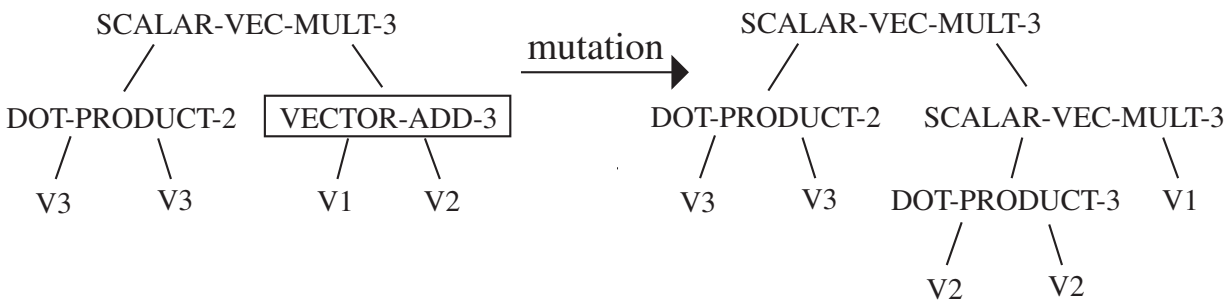

Figure 8: Mutation for STGP.

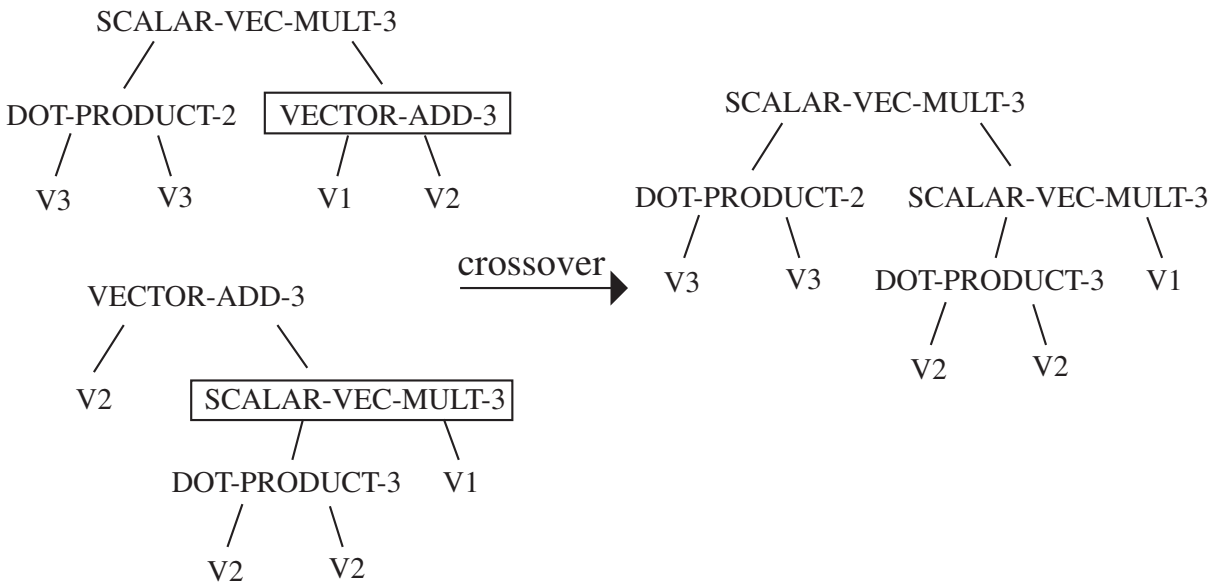

Figure 9: Crossover for STGP.

Function Name	Arguments	Return Type
DOT-PRODUCT	VECTOR-i VECTOR-i	FLOAT
VECTOR-ADD	VECTOR-i VECTOR-i	VECTOR-i
MAT-VEC-MULT	MATRIX-i-j VECTOR-j	VECTOR-i
CAR	LIST-OF-t	t
LENGTH	LIST-OF-t	INTEGER
IF-THEN-ELSE	BOOLEAN t t	t

Figure 10: Some generic functions with their argument types and return types. Here, i and j are arbitrary integers and t is an arbitrary data type.

2.2 Generic Functions

The examples above illustrate a major inconvenience of the basic STGP formulation, the need to specify multiple functions which perform the same operation on different types. For example, it is inconvenient to have to specify both DOT-PRODUCT-2 and DOT-PRODUCT-3 instead of a single function DOT-PRODUCT. To eliminate this inconvenience, we introduce the concept of a "generic function". A generic function is a function which can take a variety of different argument types and, in general, return values of a variety of different types. The only constraint is that for any particular set of argument types a generic function must return a value of a well-defined type. Specifying a set of argument types (and hence also the return type) for a generic function is called "instantiating" the generic function.

Some examples of generic functions are shown in Figure 10. Note how in each case specifying the argument types precisely allows one to deduce the return type precisely. For example, specifying that CAR's argument is of type LIST-OF-FLOAT implies that its returned value is of type FLOAT, or specifying that MAT-VEC-MULT's arguments are of type MATRIX-3-2 and VECTOR-2 implies that its returned value is of type VECTOR-3.

To be in a parse tree, a generic function must be instantiated. Once instantiated, an instance of a generic function keeps the same argument types even when passed from parent to child. Hence, an instantiated generic function acts exactly like a standard strongly typed function. A generic function gets instantiated during the process of generating parse trees (for either initialization or mutation). Note that there can be multiple instantiations of a generic function in a single parse tree.

Because generic functions act like standard strongly typed functions once instantiated, the only changes to the STGP algorithm needed to accomodate generic functions are for the tree generation procedure. There

are three such changes required.

First, during the process of generating the types possibilities tables, recall that for standard non-terminal functions we needed just check that each of its argument types was in the table entry for depth $i-1$ in order to add its return type to the table entry for depth i. This does not work for generic functions because each generic function has a variety of different argument types and return types. For generic functions, this step is replaced with the following:

```
loop over all ways to combine the types from table_entry( i-1 ) into
      sets of argument types for the function
    if the set of arguments types is legal
       and the return type for this set of argument types is not in
             table_entry( i )
        then add the return type to table_entry( i );
end loop;
```

The second change is during the tree generation process. Recall that for standard functions, when deciding whether a particular function could be child to an existing node, we could independently check whether it returns the right type and whether its argument types can be generated. However, for generic functions we must replace these two tests with the following single test:

```
loop over all ways to combine the types from table_entry( i-1 ) into
      sets of argument types for the function
    if the set of arguments types is legal
       and the return type for this set of argument types is correct
        then return that this function is legal;
end loop;
return that this function is not legal;
```

The third change is also for the tree generation process. Note that there are two types of generic functions, ones whose argument types are fully determined by selection of their return types and ones whose argument types are not fully determined by their return types. We call the latter "generic functions with free arguments". Some examples of generic functions with free arguments are DOT-PRODUCT and MAT-VEC-MULT, while some examples of generic functions without free arguments are VECTOR-ADD and SCALAR-VEC-MULT. When we select a generic function with free arguments to be a node in a tree, its return type is determined by its parent node (or if it is at the root position, by the required tree type), but this does not fully specify its argument types. Therefore, to determine its arguments types and hence the return types of its children nodes, we must use the types possibilities table to determine all the possible sets of argument types which give rise to the determined return type (there must be at least one such set for this function to have been selected) and randomly select one of these sets.

Example 5 Using generic functions, we can rewrite the non-terminal set from Example 1 in a more compact form: $\mathcal{N}$ = {DOT-PRODUCT, VECTOR-ADD, SCALAR-VEC-MULT}. Recall that $\mathcal{T}$ = {V1,

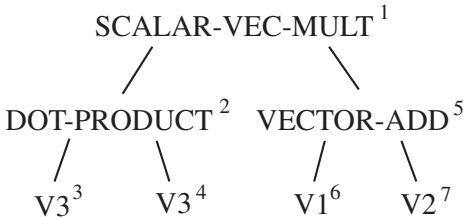

Figure 11: A legal tree using generic functions.

V2, V3}, where V1 and V2 are type VECTOR-3, and V3 is type VECTOR-2. The types possibilities tables are still as in Example 3. Figure 11 shows the equivalent of the tree in Figure 6. To generate the tree shown in Figure 11 as an example of a full tree of depth 3, we go through the following steps. At point 1, we can select either VECTOR-ADD or SCALAR-VEC-MULT, and we choose SCALAR-VEC-MULT. At point 2, we must select DOT-PRODUCT. Because DOT-PRODUCT has free arguments, we must select its argument types. Examining the types possibilities table, we see that the pairs (VECTOR-2, VECTOR-2) and (VECTOR-3, VECTOR-3) are both legal. We randomly select (VECTOR-2, VECTOR-2). Then, points 3 and 4 must be of type VECTOR-2 and hence must be V3. Point 5 must be VECTOR-ADD. (SCALAR-VEC-MULT is illegal because FLOAT is not in the types possibilities table entry for depth 1.) Points 6 and 7 can both be either V1 or V2, and we choose V1 for point 6 and V2 for point 7.

2.3 Generic Data Types

A generic data type is not a true data type but rather a set of possible data types. Examples of generic data types are VECTOR-GENNUM1, which represents a vector of arbitrary dimension, and GENTYPE2, which represents an arbitrary type. Generic data types are treated differently during tree generation than during tree execution. When generating new parse trees (either while initializing the population or during reproduction), the quantities such as GENNUM1 and GENTYPE2 are treated like algebraic quantites. Examples of how generic data types are manipulated during tree generation are shown in Figure 12. During execution, the quantities such as GENNUM1 and GENTYPE2 are given specific values based on the data used for evaluation. For example, if in the evaluation data, we choose a particular vector of type VECTOR-GENNUM1 to be a two-dimensional vector, then GENNUM1 is equal to two for the purpose of executing on this data. The following two examples illustrate generic data types.

Example 6 Consider the same terminal set $\mathcal{T}$ = {V1, V2, V3} and non-terminal set $\mathcal{N}$ = {DOT-PRODUCT, VECTOR-ADD, SCALAR-VEC-MULT} as in Example 5. However, we now set V1 and V2 to be of type VECTOR-GENNUM1, V3 to be of type VECTOR-GENNUM2, and the return type for the tree to be VECTOR-GENNUM1. It is still illegal to add V1 and V3 because they are of different dimensions, while it is still legal to add V1 and V2 because they are of the same dimension. In fact, the set of all legal parse trees for Example 5 is the same as the set of legal parse trees for this example. The difference is that in Example 5, when providing data for the evaluation function, we were constrained to have V1 and V2 of dimension 3 and V3 of dimension 2. In this example, when providing examples, V1, V2 and V3 can be arbitrary vectors as long as V1 and V2 are of the same dimension.

Function	Arguments	Return Type
VECTOR-ADD	VECTOR-GENNUM2 VECTOR-GENNUM2	VECTOR-GENNUM2
VECTOR-ADD	VECTOR-GENNUM2 VECTOR-3	illegal different dimensions
VECTOR-ADD	VECTOR-GENNUM2 VECTOR-GENNUM1	illegal different dimensions
VECTOR-ADD	GENTYPE1 GENTYPE1	illegal not vectors
CAR	LIST-OF-GENTYPE3	GENTYPE3
CAR	GENTYPE3	illegal, not a list
IF-THEN-ELSE	BOOLEAN GENTYPE3 GENTYPE3	GENTYPE3

Figure 12: Some examples of how generic data types are manipulated.

Example 7 Consider the same terminal and non-terminal sets as Examples 5 and 6. However, we now specify that V1 is of type VECTOR-GENNUM1, V2 is of type VECTOR-GENNUM2, and V3 is of type VECTOR-GENNUM3. Now, it is not only illegal to add V1 and V3, but it is also illegal to add V1 and V2, even if V1 and V2 both happen to be of type VECTOR-3 (i.e., GENNUM1 = GENNUM2 = 3) in the data provided for the evaluation function. In fact, the majority of the trees legal for Examples 5 and 6 are illegal for this example, including that in Figure 11.

One reason to use generic data types is to eliminate operations which are legal for a particular set of data used to evaluate performance but which are illegal for other potential sets of data. Some examples are the function NTH discussed in Section 3.4 and the function MAPCAR discussed in Section 3.5. For the evaluation functions, we only use lists of type LIST-OF-FLOAT. Without generic data types, we can perform operations such as (+ (CAR L) (CAR L)). However, both NTH and MAPCAR should work on any list including lists of types such as LIST-OF-STRING and LIST-OF-LIST-OF-FLOAT, and hence the expression (+ (CAR L) (CAR L)) should be illegal. With generic data types, for the purpose of generating trees, the lists are of type LIST-OF-GENTYPE1, and this expression is illegal.

Another advantage of using generic data types is that by eliminating certain operations it reduces the size of the search space, often by a large factor. Smaller search spaces mean less time required for the search. With noisy evaluation functions, smaller search spaces also help with the problem of "overfitting," i.e. fitting the solution to model the random noise in the evaluation data in addition to the "signal" (which is a problem which arises in the Kalman filter example described in Section 3.3).

A third reason to use generic data types is that, when using generic data types, the functions that are learned during genetic programming are generic functions. To see what this means, note that in each of

Function	Arguments	Returns
SET-VARIABLE-2	variable 2's type	VOID
TURN-RIGHT	FLOAT	VOID
EXECUTE-TWO	VOID t	t
DOTIMES	INTEGER VOID	VOID

Figure 13: Some examples of functions which use the VOID data type.

the examples of Section 3, we are learning a function which, like any other STGP function takes typed arguments and returns a typed value. For example, NTH is a function which takes as arguments a list and an integer and returns a value whose type is that of the elements of the list, while GENERALIZED-LEAST-SQUARES is a function which takes a matrix and a vector as arguments and returns a vector. Without generic data types, these functions STGP learns are non-generic functions, taking fully specified data types for arguments and returning a value of a fully specified data type. (For example, NTH might take a LIST-OF-FLOAT and an INTEGER as arguments and return a FLOAT.) However, all these functions should instead be generic functions (e.g., NTH should take a LIST-OF-t and an INTEGER and return a t, where t is an arbitrary type), and using generic data types makes this possible. This becomes particularly important when we start using the learned functions as building blocks for higher-level functions.

2.4 The VOID Data Type

There is a special data type which indicates that a particular subroutine is a procedure rather than a function, i.e. returns no data. We call this data type "VOID" to be consistent with C and C++, which use this same special data type for the same purpose [Kernighan and Ritchie 78]. Such procedures act only via their side effects, i.e. by changing some internal state.

Some examples of functions that have arguments and/or returned values of type VOID are shown in Figure 13. SET-VARIABLE-2 has the effect of changing a local variable's value (see Section 2.5). TURN-RIGHT has the effect of making some agent such as a robot or a simulated ant turn to the right a certain angle. EXECUTE-TWO executes two subtrees in sequence and returns the value from the second one. DOTIMES executes its second subtree a certain number of times in succession, with the number of executions determined by the value returned from the first subtree.

Instead of having EXECUTE-TWO and DOTIMES both take VOIDs as arguments, we could have had them take arbitrary types as arguments. The philosophy behind our choice to use VOID's is that the values of these arguments are never used; hence, the only effect of these arguments are their side effects. While a function may both return a value and have side effects, if it is ever useful to just execute the side effects of this function then there should be a separate procedure which just executes these side effects. Eliminating from parse trees the meaningless operations involved with computing values and then not using them reduces the size of the search space.

Type	Value
FLOAT	0.0
INTEGER	0
LIST-OF-t	empty list
VECTOR-i	all entries 0.0

Figure 14: Default values for different variable types.

Additionally, generic functions which handle arbitrary types can also handle type VOID. For example, IF-THEN-ELSE can take VOID's as its second and third arguments and return a VOID.

2.5 Local Variables

Most high-level programming languages provide local variables, which are slots where data can be stored during the execution of a subroutine. STGP also provides local variables. Like the terminal and non-terminal sets, the local variables and their types have to be specified by the user. For example, the user might specify that variable 1 has type VECTOR-GENNUM1 and variable 2 has type INTEGER.

For any local variable i, there are two functions automatically defined: SET-VAR-i takes one argument whose type is the same as that of variable i and returns type VOID, while GET-VAR-i takes no arguments and returns type the same as that of variable i. In fact, the only effect of specifying a local variable is to define these two functions and add GET-VAR-i to the terminal set and SET-VAR-i to the non-terminal set. SET-VAR-i sets the value of variable i equal to the value returned from the argument. GET-VAR-i returns the value of variable i, which is the last value it was set to or, if variable i has not yet been set, is the default value for variable i's type. Figure 14 shows some of the default values we have chosen for different types.

2.6 Run-Time Errors

STGP avoids one important type of error, that of mismatched types, by using strong typing to ensure that all types are consistent. However, there are other types of errors which occur when executing a parse tree, which we call "run-time errors". Our implementation of STGP handles run-time errors as follows. Functions which return values (i.e., non-VOID functions) always return pointers to the data. When a function gets an error, it instead returns a NULL pointer. Functions which return type VOID, i.e. procedures, also signal errors by returning a NULL pointer and signal successful operation by returning an arbitrary non-NULL pointer. When one of the arguments of a function returns a NULL pointer, this function stops executing and returns a NULL pointer. In this way, errors get passed up the tree.

The function which initially detects the error sets a global variable (analogous to the UNIX global variable "errno") to indicate the type of error. The reason the function needs to specify the type of error is so that the evaluation function can use this information. For example, in the unprotected version of the NTH function (see Section 3.4), when the argument N specifies an element beyond the end of the list, then a

Bad-List-Element error (see below) is the right response but a Too-Much-Time error is a bad response. Eventually, we would also like to make the error type available to functions in the tree. For example, we could define a function ERROR-TYPE which returns the type of error and use it it as part of a terminal set.

The current error types are:

Inversion-Of-Singular-Matrix: The MATRIX-INVERSE function performs a Gaussian elimination procedure. During this procedure, if a column has entries with absolute values all less than some very small value ϵ, then the inversion fails with this error type.

Note that this is a very common error when doing matrix manipulation because it is easy to generate square matrices with rank less than their dimension. For example, if A is an mxn matrix with $m < n$, then A^TA is an nxn matrix with rank $\leq m$ and hence is singular. Likewise, if $m > n$, AA^T is an mxm matrix with rank $\leq n$ and hence is singular. Furthermore, $(AA^T)^{-1}A$ and $A(A^TA)^{-1}$ have the same dimension as A and hence can be used in trees any place A can (disregarding limitations on maximum depth).

Also note that at one point we used a protected inversion which, analogous to Koza's protected division, returned the identity matrix when attempting to invert a singular matrix. However, this had two problems. First, when all the examples in the evaluation data yield a singular matrix, then a protected inversion of this matrix generally yields incorrect results for cases when the matrix is nonsingular. For example, in the evaluation data of the least squares example (see Section 3.2), we chose A to have dimensions 20x3. Optimizing to this value of A yields expressions with extra multiplications by $(AA^T)^{-1}$ included. The problem is that this expression is also supposed to be optimal when A is a square matrix and hence AA^T is (generally) invertible, which is not the case with these extra multiplications included. The second problem is that, when all the examples in the evaluation data yield a nonsingular matrix, then a protected inversion of this matrix generally yields incorrect results for cases when the matrix is singular. As an example, again consider the least squares problem. When A^TA is singular, then there are multiple optimal solutions. The right thing in this case may be to return one of these solutions or may be to raise an error, but a protected inversion will do neither of these.

Bad-List-Element: Consider taking the CAR (i.e., first element) of an empty list. In Lisp, taking the CAR of NIL (which is the empty list) will return NIL. The problem with this for STGP is that of type consistency; CAR must return data of the same type as the elements of the list (e.g., must return a FLOAT if the list is of type LIST-OF-FLOAT). There are two alternative ways to handle this: first, raise an error, and second, have a default value for each possible type. For reasons similar to those given for not using protected matrix inversion, we choose to return an error rather than have a protected CAR.

Note that CAR is not the only function which can have this type of error. For example, the unprotected version of the function NTH (see Section 3.4) raises this error when the argument N is $\geq$ the length of the argument L.

Division-By-Zero: We do not use scalar division in any of our examples and hence do not use this error type. We include this just to show that there is an alternative to the protected division used by Koza, which returns 1 whenever division by zero is attempted.

Too-Much-Time: Certain trees can take a long time to execute and hence to evaluate, particularly

those trees with many nested levels of iteration. To ensure that the evaluation does not get bogged down evaluating one individual, we place a problem-dependent limit on the maximum amount of time allowed for an evaluation. Certain functions check whether this amount of time has elapsed and, if so, raise this error. Currently, DOTIMES and MATRIX-INVERSE are the only functions which perform this check. DOTIMES does this check before each time executing the loop body, while MATRIX-INVERSE does this check before performing the inversion.

2.7 STGP's Programming Language

In the process of defining STGP, we have taken the first steps towards defining a new programming language. This language is a cross between Ada and Lisp. The essential ingredient it takes from Ada is the concept of strong typing and the concept of generics as a way of making strongly typed data and functions practical to use [Barnes 82]. The essential ingredient it takes from Lisp is the concept of having programs basically be their parse trees [Steele 84]. The resulting language might best be considered a strongly typed Lisp. [Note that it is important here to distinguish between a language and its parser. While the underlying learning mechanism (the analog of a compiler) for standard genetic programming can be written in any language, the programs being learned are Lisp programs. Similarly, while the learning mechanism for STGP can be written in any language, the programs being manipulated are in this hybrid language.]

There are reasons why a strongly typed Lisp is a good language not only for learning programs using genetic algorithms but also for any type of automatic learning of programs. Having the programs be isomorphic to their parse trees makes the programs easy to create, revise and recombine. This not only makes it easy to define genetic operators to generate new programs but also to define other methods of generating programs. Strong typing makes it easy to ensure that the automatically generated programs are actually legal programs. The standard dynamically typed Lisp relies on the abilities of human programmers to ensure that all data types are manipulated legally. This in turn relies on the human programmers to "understand" the functions they call and/or to "read the documentation". A more formal approach is required to allow automatic programming techniques, which often rely on large amounts of computation rather than understanding, to generate legal trees. The ability to define such a formal approach is provided by strong typing.

3 Experiments

We now discuss four problems to which we have applied STGP. Multi-dimensional least squares regression (Section 3.2) and the multi-dimensional Kalman filter (Section 3.3) are two problems involving vector and matrix manipulation. The function NTH (Section 3.4) and the function MAPCAR (Section 3.5) are two problems involving list manipulation. However, before discussing these four experiments, we first describe the genetic algorithm we used for these experiments.

3.1 The Genetic Algorithm

The genetic algorithm we used differs from a "standard" genetic algorithm in some ways which are not due to the use of trees rather than strings as chromosomes. We now describe these differences so that readers can best analyze and (if desired) reproduce the results.

The code used for this genetic algorithm is a C++ translation of an early version of OOGA [Davis 91]. One important distinction of this genetic algorithm is its use of steady-state replacement [Syswerda 89] rather than generational replacement for performing population updates. This means that for each generation only one individual (or a small number of individuals) is generated and placed in the population rather than generating a whole new population. The benefit of steady-state replacement is that good individuals are immediately available as parents during reproducation rather than waiting to use them until the rest of the population has been evaluated, hence speeding up the progress of the genetic algorithm. When using steady-state replacement, it does not make sense to report run durations as a number of generations but rather as the total number of evaluations performed. Comparisons of results between different genetic algorithms should be made in units of number of evaluations. (Since steady-state genetic algorithms can be parallelized [Montana 91], such a comparison is fair.)

A second important features of this code is the use of exponential fitness normalization [Cox, Davis and Qiu 91]. This means that, when selecting parents for reproduction, (i) the probability of selecting any individual depends only on its relative rank in the population and (ii) the probability of selecting the n^{th} best individual is Parent-Scalar times that of selecting the $(n-1)^{st}$ best individual. Here, Parent-Scalar is a parameter of the genetic algorithm which must be < 1. An important benefits of exponential fitness normalization is the ability to simply and precisely control the rate of convergence via the choice of values for the population size and the parent scalar. Controlling the convergence rate is important to avoid both excessively long runs and runs which converge prematurely to a non-global optimum. The effect of the population size and the parent scalar on convergence rate is detailed in [Montana 94]. For the purposes of this paper, it is enough to note that increasing the population size and increasing the parent scalar each slow down the convergence rate.

3.2 Multi-Dimensional Least Squares Regression

Problem Description: The multi-dimensional least squares regression problem can be stated as follows. For an mxn matrix A with $m \geq n$ and an m-vector B, find the n-vector X which minimizes the quantity $(AX - B)^2$. This problem is known to have the solution

$$X = (A^T A)^{-1} A^T B \tag{1}$$

where $(A^T A)^{-1} A^T$ is called the "pseudo-inverse" of A [Campbell and Meyer 1979]. Note that this is a generalization of the linear regression problem, given m pairs of data (x_i, y_i), find m and b such that the line $y = mx + b$ gives the best least-squares fit to the data. For this special case,

$$A = \begin{bmatrix} x_1 & 1 \\ ... & ... \\ x_m & 1 \end{bmatrix} \quad B = \begin{bmatrix} y_1 \\ ... \\ y_m \end{bmatrix} \quad X = \begin{bmatrix} m \\ b \end{bmatrix} \tag{2}$$

Output Type: The output has type VECTOR-GENNUM1.

Arguments: The argument A has type MATRIX-GENNUM2-GENNUM1, and the argument B has type VECTOR-GENNUM2.

Local Variables: There are no local variables.

Terminal Set: $\mathcal{T} = \{\text{A}, \text{B}\}$

Non-Terminal Set: We use two non-terminal sets

$$\mathcal{N}_1 = \{\text{MATRIX_TRANSPOSE}, \text{MATRIX_INVERSE}, \text{MAT_VEC_MULT}, \text{MAT_MAT_MULT}\} \quad (3)$$

$$\mathcal{N}_2 = \{\text{MATRIX_TRANSPOSE}, \text{MATRIX_INVERSE}, \text{MAT_VEC_MULT}, \text{MAT_MAT_MULT}, \text{MATRIX_ADD}, \text{MATRIX_SUBTRACT}, \text{VECTOR_ADD}, \text{VECTOR_SUBTRACT}, \text{DOT_PRODUCT}, \text{SCALAR_VEC_MULT}\} \quad (4)$$

The first is the minimal non-terminal set necessary to solve the problem.

Evaluation Function: We used a single data point for the evaluation function. Because this is a deterministic problem which has a solution which does not have multiple cases, a single data point is in theory all that is required. For this data point, we chose GENNUM1=3 and GENNUM2=20, so that A was a 20x3 matrix and B was a 20-vector. The entries of A and B were selected randomly. The score for a particular tree was $(AX - B)^2$, where X is the 3-vector obtained by executing the tree.

Genetic Parameters: We chose MAX-INITIAL-DEPTH to be 6 and POPULATION-SIZE to be 50 and 2000 respectively for the two different non-terminal sets. Other parameters were irrelevant because an optimal solution was always found in the initial population.

Results: We ran STGP ten times with non-terminal set $\mathcal{N}_1$ (and population size 50) and ten times with non-terminal set $\mathcal{N}_2$ (and population size 2000). Every time at least one optimal soluation was found as part of the initial population. With $\mathcal{N}_1$, there was an average of 2.9 optimal parse trees in the initial population of 50. Of the 29 total optimal trees generated over 10 runs, there were 14 distinct optimal trees. (Note that because we are using a steady-state genetic algorithm there can be no duplicate trees in any one run.) In the second case, there was an average of 4.6 optimal trees in the initial population of 2000.

We now look at a sampling of some of these optimal parse trees The two with the minimum number of nodes (written as S-expressions) are:

```
(1) (MAT-VEC-MULT (MATRIX-INVERSE (MAT-MAT-MULT (MATRIX-TRANSPOSE A) A))
                  (MAT-VEC-MULT (MATRIX-TRANSPOSE A) B))
(2) (MAT-VEC-MULT (MAT-MAT-MULT
                       (MATRIX-INVERSE (MAT-MAT-MULT (MATRIX-TRANSPOSE A) A))
                       (MATRIX-TRANSPOSE A))
                  B)
```

Tree 1, in addition to having the minimum number of nodes, also has the minimum depth, 5. It is the only

Max Depth	Legal Trees	GP Trees	Fraction Legal
1	0	2	0.0
2	0	38	0.0
3	3	11630	2.6e-4
4	164	1.1e9	1.5e-7
5	2.9e5	9.4e18	3.1e-14
6	8.9e11	7.0e38	1.3e-27

(a) Grow

Max Depth	Legal Trees	GP Trees	Fraction Legal
1	0	2	0.0
2	0	36	0.0
3	2	10440	1.9e-4
4	60	8.7e8	6.9e-8
5	32406	6.1e18	5.3e-15
6	8.0e9	3.0e38	2.7e-29

(b) Full

Figure 15: Legal trees vs. GP trees for different tree sizes using the non-terminal set $\mathcal{N}_2$.

optimal tree of depth 5 when using the non-terminal set $\mathcal{N}_1$. However, with $\mathcal{N}_2$, there are many optimal trees of depth 5 including

```
(1) (MAT-VEC-MULT (MATRIX-INVERSE (MAT-MAT-MULT (MATRIX-TRANSPOSE A)
                                                (MATRIX-ADD A A)))
                  (MAT-VEC-MULT (MATRIX-TRANSPOSE A) (VECTOR-ADD B B)))
```

To see how important strong typing was in this problem, we did another small experiment. Using non-terminal set $\mathcal{N}_2$, we counted the number of legal trees (i.e., those trees whose types are consistent) of a given size as well as the number of GP trees (i.e., those trees whose nodes all have the right number of subtrees/arguments but which are not necessarily type consistent) of this size. We then calculated the ratio of legal trees to GP trees to determine the probability of a tree of this size randomly selected by a non-strongly-typed genetic programming algorithm being a legal tree. The results for different sizes are shown in Figure 15.

Analysis: While this problem is too easy to exercise the GP (genetic programming) part of STGP (the three problems discussed below do exercise it), it clearly illustrates the importance of the ST (strongly typed) part. What the results of Figure 15 show is that just generating legal parse trees is a computationally intractable task if we try to do this by generating random GP trees and then selecting out the legal ones. Our method of generating only legal parse trees is clearly a superior approach and is what makes a dauntingly difficult problem into an easy one.

3.3 The Kalman Filter

Problem Description: The Kalman filter is a popular method for tracking the state of a system with stochastic behavior using noisy measurements [Kalman 60]. A standard formulation of a Kalman filter is the following. Assume that the system follows the stochastic equation

$$\dot{\vec{x}} = A\vec{x} + B\vec{n}_1 \tag{5}$$

where $\vec{x}$ is an n-dimensional state vector, A is an nxn matrix, $\vec{n}_1$ is an m-dimensional noise vector, and B is an nxm matrix. We assume that the noise is Gaussian distributed with mean 0 and covariance the mxm matrix Q. Assume that we also make continuous measurements of the system given by the equation

$$\vec{y} = C\vec{x} + \vec{n}_2 \tag{6}$$

where $\vec{y}$ is a k-dimensional output (or measurement) vector, C is a kxn matrix, and $\vec{n}_2$ is a k-dimensional noise vector which is Gaussian distributed with mean 0 and covariance the kxk matrix R. Then, the estimate $\hat{\vec{x}}$ for the state which minimizes the sum of the squares of the estimation errors is given by

$$\dot{\hat{\vec{x}}} = A\hat{\vec{x}} + PC^T R^{-1}(\vec{y} - C\hat{\vec{x}}) \tag{7}$$

$$\dot{P} = AP + PA^T - PC^T R^{-1} CP + BQB^T \tag{8}$$

where P is the covariance of the state estimate.

The work that we have done so far has focused on learning the right-hand side of Equation 7. In the "Analysis" portion, we discuss why we have focused on just the state estimate part (i.e., Equation 7) and what it would take to simultaneously learn the covariance part (i.e., Equation 8).

Output Type: The output has type VECTOR-GENNUM1.

Arguments: The arguments are: A has type MATRIX-GENNUM1-GENNUM1, C has type MATRIX-GENNUM2-GENNUM1, R has type MATRIX-GENNUM2-GENNUM2, P has type MATRIX-GENNUM1-GENNUM1, Y has type VECTOR-GENNUM2, and X-EST has type VECTOR-GENNUM1.

Local Variables: There are no local variables.

Terminal Set: $\mathcal{T} = \{\text{A}, \text{C}, \text{R}, \text{P}, \text{Y}, \text{X_EST}\}$

Non-Terminal Set:

$$\begin{aligned} \mathcal{N} = \{&\text{MAT_MAT_MULT}, \text{MATRIX_INVERSE}, \text{MATRIX_TRANSPOSE} \\ &\text{MAT_VEC_MULT}, \text{VECTOR_ADD}, \text{VECTOR_SUBTRACT}\} \end{aligned} \tag{9}$$

Evaluation Function: Before running the genetic algorithm, we created a track using Equations 5 and 6 with the parameters chosen to be

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, C = \begin{bmatrix} 1 & 2 & -1 \\ 3 & -1 & 1 \end{bmatrix} \tag{10}$$

$$B = \begin{bmatrix} 1 & -1 \\ 2 & 1 \\ 0 & -3 \end{bmatrix}, Q = \begin{bmatrix} 2.5 & -0.25 \\ -0.25 & 1.25 \end{bmatrix}, R = \begin{bmatrix} 0.05 & 0 \\ 0 & 0.05 \end{bmatrix} \tag{11}$$

Note that this choice of parameters implies GENNUM1=3 and GENNUM2=2. We used a time step of $\delta t = 0.005$ (significantly larger time steps caused unacceptably large approximation errors, while significantly smaller time steps caused unacceptably large computation time in the evaluation function) and ran the

system until time $t = 4$ (i.e., for 800 time steps), recording the values of $\vec{x}$ and $\vec{y}$ after each time step. The initial conditions of the track were

$$\vec{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, P = 0 \tag{12}$$

The genetic algorithm reads in and stores this track as part of its initialization procedure.

Evaluating a parse tree is done as follows. Start with $\hat{\vec{x}}$ equal to the initial value of $\vec{x}$ given in Equation 12 and with $P = 0$. For each of the 800 time steps, update P and $\hat{\vec{x}}$ according to

$$\hat{\vec{x}}_{new} = \hat{\vec{x}}_{old} + \delta t * (\text{value returned by tree}), P_{new} = P_{old} + \delta t * (\text{right}_{\text{h}}\text{and}_{\text{s}}\text{ide of Equation 8}) \tag{13}$$

After each time step, compute the difference between the state estimate and the state for that step. The score for a tree is the sum over all time steps of the square of this difference.

Note that there is no guarantee that a parse tree implementing the correct solution given in Equation 7 will actually give the best score of any tree. Two possible sources of variation are (i) the quantization effect introduced by the fact that the step size is not infinitesimally small and (ii) the stochastic effect introduced by the fact that the number of steps is not infinitely large. This is the problem of overfitting which we briefly discussed in Section 2.3.

Genetic Parameters: We used a population size of 50,000, a parent scalar of 0.9995, a maximum initial depth of 5, and a maximum depth of 7.

Results: We ran STGP 10 times with the specified parameters, and each time we found the optimal solution given in Equation 7. To find the optimal solution required an average of 92,800 evaluations and a maximum of 117,200 evaluations. A minimal parse tree implementing this solution is

```
(VECTOR-ADD (MAT-VEC-MULT A X)
         (MAT-VEC-MULT (MAT-MAT-MULT P (MATRIX-INVERSE R))
                   (MAT-VEC-MULT (MATRIX-TRANSPOSE C)
                             (VECTOR-SUBTRACT Y (MAT-VEC-MULT C X)))))
```

On each of the runs, we allowed the genetic algorithm to execute for some number of evaluations after finding the optimal solution, generally around 5000 to 10000 extra evaluations. In one case, STGP found a "better" solution than the "optimal" one, i.e. a parse tree which gave a better score than the solution given in Equation 7. Letting this run continue eventually yielded a "best" tree which implemented the solution

$$\dot{\hat{\vec{x}}} = A\hat{\vec{x}} + PC^T R^{-1}(\vec{y} - C\hat{\vec{x}}) - P^2\hat{\vec{x}} \tag{14}$$

and which had a score of 0.00357 as compared with a score of 0.00458 for trees which implemented Equation 7.

Analysis: Given appropriate genetic parameters and a sufficient number of evaluations, STGP had no trouble finding the theoretically optimal solution for the state estimate given the correct covariance. This is a validation of STGP's ability to solve a moderately difficult problem in vector/matrix manipulation

In one run, STGP was able to overfit to the training data by finding an additional term which modeled the noise. Because the search space is finite, there exists some time step small enough and some total time large enough so that the theoretically optimal solution will also be the one which evaluates best on the data. However, due to lack of computational power, we were not able to experiment with different time steps and total times in order to find out what time step and total time are required to prevent overfitting. (The evaluation time for a parse tree is proportional to the number of data points in the generated track.)

One indication of STGP's current shortcomings was the fact that we were not able to make a serious attempt at solving the full problem, i.e. simultaneously deriving the state estimate (Equation 7) and the covariance (Equation 8). The problem is twofold: first, the combined expression is much more complicated than the single vector expression, and second, the terms in the covariance update equation have only higher-order effects on the state estimate and hence the error. This means that large amounts of data are needed to allow these higher-order effects to not be washed out by noise. The combined effect is that the required computational power is far beyond what we had available to us for these experiments.

3.4 The Function NTH

Problem Descrition The Lisp function NTH takes two arguments, a list L and an integer N, and returns the N^{th} element of L. The standard definition of NTH [Steele 84] specifies that it actually returns the $(N+1)^{st}$ element of the list; e.g., for $N = 1$, NTH will return the second element of the list. For $N < 0$, NTH is defined to raise an error, and for $N \geq \text{length}(L)$, NTH is defined to return NIL, the empty list.

Using STGP, we cannot learn NTH defined this way because of the type inconsistency caused by returning the empty list in some situations. Instead, we define three variations on the function NTH, which in increasing order of complexity are

- NTH-1 is identical to NTH except that for $N \geq \text{length}(L)$ it raises a Bad-List-Element error instead of returning the empty list and for $N < 0$ it returns the first element rather than raising an error. (The latter change is just for simplicity.)
- NTH-2 is the same as NTH-1 except that it actually return the N^{th} element of L rather than the $(N+1)^{st}$ element.
- NTH-3 is the same as NTH-2 except that for $N > \text{length}(L)$ it returns the last element of the list instead of raising a Bad-List-Element error.

Output Type: The output has type GENTYPE1.

Arguments: The argument N has type INTEGER, and the argument L has type LIST-OF-GENTYPE1.

Local Variables: Variable 1 has type LIST-OF-GENTYPE1.

Terminal Set: $\mathcal{T} = \{\text{N}, \text{L}, \text{GET_VAR_1}\}$

Non-Terminal Set: We used three different non-terminal sets for the three variations of NTH. For

NTH-1, NTH-2 and NTH-3 respectively, they are

$$\mathcal{N}_1 = \{\text{CAR}, \text{CDR}, \text{EXECUTE_TWO}, \text{DOTIMES}, \text{SET_VAR_1}\} \quad (15)$$

$$\mathcal{N}_2 = \{\text{CAR}, \text{CDR}, \text{EXECUTE_TWO}, \text{DOTIMES}, \text{SET_VAR_1}, \text{ONE}, \text{PLUS}, \text{MINUS}\} \quad (16)$$

$$\mathcal{N}_3 = \{\text{CAR}, \text{CDR}, \text{EXECUTE_TWO}, \text{DOTIMES}, \text{SET_VAR_1}, \text{ONE}, \text{PLUS}, \text{MINUS}, \text{MIN}, \text{LENGTH}\} \quad (17)$$

Evaluation Function: For NTH-1, we used 53 different examples to evaluate performance. Each example had N assume a different value in the range from -1 to 51. For all the examples we took L to be a list of length 50 with all of its entries unique. For each example, the evaluation function executed the tree and compared the returned value with the expected result to compute a score; the scores for each example were then summed into a total score. The score for each example was defined as 0 if the correct behavior was to raise an error but the tree returned a value, 0 if the correct behavior was to raise an error but the tree returned a value, 10 if the correct behavior was to raise an error and the tree raised an error, and $10 * 2^{-d}$ if the correct behavior was to return a value and the tree returned a value that was d positions away from the correct position in the list. For example, if the list was $(3, 1, 9, 4, ...)$ and N was 3, then a tree that returned 9 would get a score of 5 for this example while a tree that returned 3 would get a score of 1.25.

For NTH-2, we used the same evaluation function as for NTH-1 except that N assumed values in the range 0 to 52 and the expected result for each example was the N^{th} rather than the $(N+1)^{st}$ list element.

For NTH-3, we used the same evaluation function as for NTH-2 with the following changes. First, there was no case in which the correct behavior was to raise an error; for the cases when $N >$ length(L), the correct behavior is to return the last list element. Second, we shortened the length of L to 20 instead of 50, purely for the purpose of speeding up the evaluation function. Third, we allowed N to range from 0 to 26; the large number of cases with $N >$ length(L) was to amply reward a tree which handled this case correctly.

Genetic Parameters: For NTH-1 and NTH-2, we used a population size of 1000, a parent scalar of 0.99, a maximum initial depth of 5, and a maximum depth of 7. For NTH-3, we used a population size of 15,000, a parent scalar of 0.9993, a maximum initial depth of 6, and a maximum depth of 8.

Results: We made ten runs of the genetic algorithm for the NTH-1 problem. All ten runs found an optimal solution with an average of 1335 trees evaluated before finding a solution. Five of the runs found an optimal solution in the initial population of 1000, and the longest run required 1900 evaluations (1000 for the initial population and 900 more for trees generated during reproduction). A tree which is minimal with respect to nodes and depth is

```
(EXECUTE-TWO (DOTIMES (EXECUTE-TWO (SET-VAR-1 L) N)
                      (SET-VAR-1 (CDR GET-VAR-1)))
             (CAR GET-VAR-1))
```

We made ten runs of the genetic algorithm for the NTH-2 problem. All ten runs found an optimal solution with an average of 2435 and a maximum of 3950 trees evaluated before finding a solution. A tree which is a solution and which is minimal with respect to nodes and depth is

```
(EXECUTE-TWO (DOTIMES (EXECUTE-TWO (SET-VAR-1 L) (- N 1))
                      (SET-VAR-1 (CDR GET-VAR-1)))
             (CAR GET-VAR-1))
```

For NTH-2, we also performed an experiment to determine the effectiveness of random search. This consisted of randomly generating parse trees using the same method used to generate the initial population of the genetic algorithm: ramped-half-and-half with a maximum depth of 5 plus a check to make sure that each tree generated is unique from all the others. However, for this experiment, we kept generating trees until we found an optimal one. The first such run of the random search algorithm required 60,200 trees to be evaluated before finding an optimal one. A second run required 49,600 trees to be evaluated.

We made ten runs of the genetic algorithm for the NTH-3 problem. Nine out of ten runs found an optimal solution with an average of 35,280 and a maximum of 44,800 trees evaluated before finding a solution. The only unsuccessful run converged to a point where the 15000 members of the population provided 15000 different solution to the NTH-2 problem. A tree which is a solution to NTH-3 and which is minimal with respect to nodes and depth is

```
(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))
```

Analysis: The NTH-1 problem, like the least-squares regression problem, was too easy to test the genetic algorithm part of STGP. However, moving from NTH-1 to NTH-2 (adding just a little bit of complexity to the problem by adding three new functions to the non-terminal set and replacing N with (- N 1) in the minimal optimal tree) made the problem sufficiently difficult to clearly illustrate the difference between random search and genetic algorithms. While the search time for the genetic algorithm increased by only a factor of two, the search time for random search increased by a factor of approximately 25. Although computational limitations kept us from moving out further along the evaluations versus problem complexity curve for random search, these results yield the same conclusion as those of [Koza 92]: that genetic search of parse tree space is superior to random search for sufficiently complex searches, and the reason is the better scaling properties of genetic algorithms.

The NTH-3 problem is more difficult than NTH-2 for a few reasons. First, a minimal-size optimal solution requires three extra nodes in the parse tree. Second, the minimal-size optimal solution has depth 7 and hence requires us to search through a space where the parse trees can have greater depth and which is hence a much bigger space. Third, there are two extra functions in the non-terminal set. This increase in difficulty is reflected in the increase in required times to find a solution.

3.5 The Function MAPCAR

Problem Description: The Lisp function MAPCAR takes two arguments, a list L and a function FUNARG, and returns the list obtained by applying FUNARG to each element of L. Here, we show how STGP can learn this function.

Note that to be able to use the function MAPCAR as an element of a non-terminal set for learning other higher-level functions requires the concept of a functional argument, i.e. the ability to pass a function (and not the result of applying a function) as an argument to another function. We have not yet implemented functional arguments, but it is possible to do so using STGP (the functional argument will have type of the form FUNCTION-RETURNING-type1-ARGUMENT-type2-type3), and we hope to have functional arguments in the future.

Output Type: The output is of type LIST-OF-GENTYPE2.

Arguments: The argument L has type LIST-OF-GENTYPE1, and the argument FUNARG is a function taking a GENTYPE1 and returning a GENTYPE2.

Local Variables: Variable 1 is of type LIST-OF-GENTYPE1, and variable 2 is of type LIST-OF-GENTYPE2.

Terminal Set: $\mathcal{T} = \{\text{L}, \text{GET_VAR_1}, \text{GET_VAR_2}\}$

Non-Terminal Set:

$$\begin{aligned}\mathcal{N} \;=\; & \{\text{CAR}, \text{CDR}, \text{EXECUTE_TWO}, \text{DOTIMES}, \text{SET_VAR_1}, \text{SET_VAR_2}, \\ & \text{LENGTH}, \text{APPEND}, \text{FUNARG}\}\end{aligned} \tag{18}$$

Evaluation Function: To evaluate performance, we used three different lists for the argument L and one function for the argument FUNARG. The three lists were: (1) the empty list, (2) a list with a single element equal to 1, and (3) a list with 50 elements whose values are the integers between 1 and 50. The function was the identity. The score S_L for each list L given that executing the parse tree either produces an error or the list L_r is

$$S_L = \begin{cases} -10 - 2 * \text{length}(L) & \text{if error} \\ -2 * |\text{length}(L) - \text{length}(L_r)| + \sum_{e \in L} 2^{-\text{dist}(e, L_r)} & \text{otherwise} \end{cases} \tag{19}$$

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 rationale for our choice of lists is as follows. The 50-element list is the primary test of performance. Doing well on this list assures a good score. The two other lists are there to penalize slightly those parse trees which do not perform correctly on short lists. An example of a parse tree which does perfectly on the long list but gets an error on the empty list is

```
(EXECUTE-TWO
   (DOTIMES
      (EXECUTE-TWO (SET-VAR-1 L) (LENGTH (CDR L)))
      (EXECUTE-TWO (SET-VAR-2 (APPEND GET-VAR-2 (FUNARG (CAR GET-VAR-1))))
                   (SET-VAR-1 (CDR GET-VAR-1))))
   (APPEND GET-VAR-2 (FUNARG (CAR GET-VAR-1))))
```

The error comes because when L is the empty list, then variable 1 is the empty list, and taking its CAR gives an error. The above parse tree would receive a score of 500 as compared to a maximum score of 510. Some other sample parse trees with their scores are the following. The parse tree

```
(APPEND GET-VAR-2 (FUNARG (CAR GET-VAR-1)))
```

receives the minimum score of -132. The parse tree

```
(APPEND GET-VAR-2 (FUNARG (CAR L)))
```

receives a score of -88. The parse tree

```
(EXECUTE-TWO
   (DOTIMES (LENGTH L) (SET-VAR-2 (APPEND GET-VAR-2 (FUNARG (CAR L)))))
   GET-VAR-2)
```

receives a score of 20. The parse tree

```
(EXECUTE-TWO
   (DOTIMES
      (EXECUTE-TWO (SET-VAR-1 L) (LENGTH (CDR L)))
      (EXECUTE-TWO (SET-VAR-1 (CDR GET-VAR-1))
                   (SET-VAR-2 (APPEND GET-VAR-2 (FUNARG (CAR GET-VAR-1))))))
   GET-VAR-2)
```

receives a score of 241. Finally, an optimal parse tree such as

```
(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)
```

receives the maximum score of 510.

Genetic Parameters: We used a population size of 50,000, a parent scalar of 0.9998, a maximum initial depth of 6, and a maximum depth of 8.

Results: We ran STGP 10 times with the specified parameters, and 8 of these 10 runs found an optimal solution. For these runs which did find an optimal solution, the average number of individuals evaluated before finding an optimal one was 204,000, while the maximum numbers of evaluations was 300,000. In the other 2 runs, STGP converged prematurely to a population consisting of 50,000 distinct parse trees all of which evaluated to 20.

Analysis: Based on the number of evaluations required to find an optimal solution, MAPCAR was clearly the most difficult problem of those discussed in this paper. To find an optimal solution with probability

> 0.95 takes on the order of 500,000 evaluations, roughly an order of magnitude more than any of the other problems. One key factor which makes this problem difficult is the existence of a suboptimal solution which is relatively easy to find and difficult to get beyond.

The large number of evaluations required to solve MAPCAR illustrates perhaps the main shortcoming of STGP. Despite the relatively good scaling of STGP (and genetic algorithms in general) with problem complexity, the amount of computation required as a function of problem complexity grows fast enough that, with today's computers, STGP can only solve relatively simple problems.

4 Conclusion

In this paper, we have introduced the concept of Strongly Typed Genetic Programming (STGP). STGP is an extension to genetic programming which fully eliminates the closure constraint necessary for standard genetic programming. It hence allows the user to define functions which take any data types as arguments and return values of any data type. We have defined for STGP the concepts of generic functions, generic data types, local variables, and errors as a way of making STGP more practical and more powerful. In the process, we have taken the first steps towards the definition of a new programming language (essentially a strongly typed Lisp) which is particularly well suited to automatic programming.

The primary experiments we have performed illustrate the effectiveness of STGP in solving a wide variety of moderately complex problems involving multiple data types. Other experiments show: (i) the importance of using strong typing for generating trees and (ii) the importance of using a genetic algorithm rather than a random search through tree space.

However, the experiments also illustrate the current shortcomings of STGP. First, it can be difficult to define good evaluation functions, even for relatively simple problems. Second, despite the fact that STGP scales well with complexity as compared with random search, truly complex problems are beyond the ability of STGP to solve in a reasonable time with any of today's computers. While the experiments show that STGP has great potential as an automatic programming tool, further improvements are necessary for it to be able to learn truly complex programs.

References

[Barnes 82] Barnes, J. 1982. *Programming in Ada.* Addison-Wesley.

[Campbell and Meyer 1979] Campbell, S.L. and Meyer, Jr., C.D. 1979. *Generalised Inverses of Linear Transformations.* Pittman.

[Cox, Davis and Qiu 91] Cox, Jr., A.L., Davis, L. and Qiu, Y. 1991. "Dynamic Anticipatory Routing in Circuit-Switched Telecommunications Networks," in [Davis 91], pp. 124–143.

[Cramer 85] Cramer, N.L. 1985. A Representation for the Adaptive Generation of Simple Sequential Programs. *Proceedings of the Fourth International Conference on Genetic Algorithms,* pp. 183–187.

[Davis 87] Davis, L. 1987. *Genetic Algorithms and Simulated Annealing.* Pittman.

[Davis 91] Davis, L. 1991. *Handbook of Genetic Algorithms,* Von Nostrand Reinhold.

[Goldberg 89] Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

[Kalman 60] Kalman, R.E. 1960. A New Approach to Linear Filtering and Prediction Problems. *Trans. ASME: J. Basic Eng.,* vol. 82, pp. 35–45.

[Kernighan and Ritchie 78] Kernighan, B. and Ritchie, D. 1978. *The C Programming Language.* Prentice-Hall.

[Koza 92] Koza, J.R. 1992. *Genetic Programming.* The MIT Press.

[Montana 91] Montana, D. 1991. "Automated Parameter Tuning for Interpretation of Synthetic Images," in [Davis 91], pp. 282-311.

[Montana 94] Montana, D. 1994. "Genetic Search of a Generalized Hough Transform Space," in preparation.

[Perkis 93] Perkis, T. 1993. "Stack-Based Genetic Programming,"

[Steele 84] Steele, G. 1984. *Common Lisp.* Digital Equipment Corporation.

[Syswerda 89] Syswerda, G. 1989. "Uniform Crossover in Genetic Algorithms," *Proc. Third International Conference on Genetic Algorithms,* pp. 2–9.