# Strongly Typed Genetic Programming

David J. Montana
Bolt Beranek and Newman Inc.
70 Fawcett Street
Cambridge, MA 02138
dmontana@bbn.com

November 20, 2002

**Abstract**

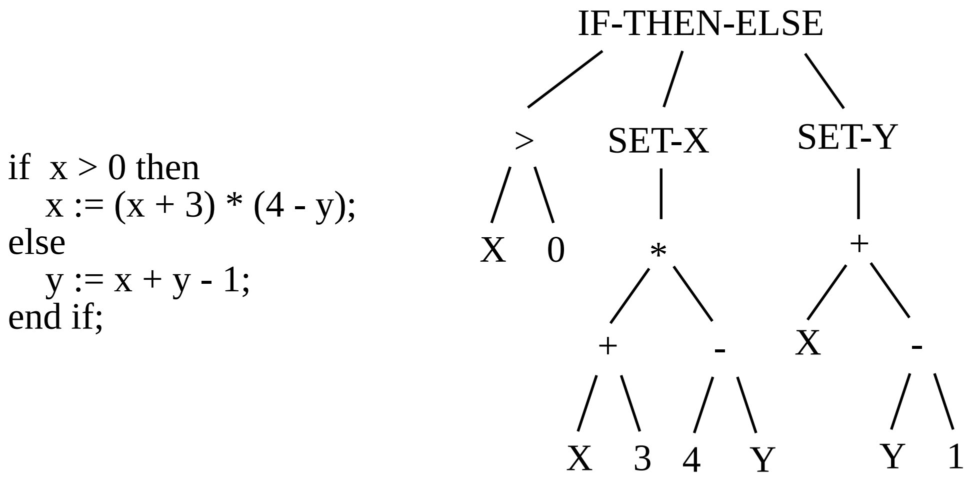
Genetic programming is a powerful method for automatically generating computer programs via the process of natural selection (Koza, 1992). However, in its standard form, there is no way to restrict the programs it generates to those where the functions operate on appropriate data types. In the case when the programs manipulate multiple data types and contain functions designed to operate on particular data types, this can lead to unnecessarily large search times and/or unnecessarily poor generalization performance. Strongly typed genetic programming (STGP) is an enhanced version of genetic programming which enforces data type constraints and whose use of generic functions and generic data types makes it more powerful than other approaches to type constraint enforcement. After describing its operation, we illustrate its use on problems in two domains, matrix/vector manipulation and list manipulation, which require its generality. The examples are: (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, 1992). 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, 1985)), it is so far the most successful. In Section 1.1 we discuss genetic programming and how it differs from a standard genetic algorithm. In Section 1.2 we examine type constraints in genetic programming: the need for type constraints, previous approaches, and why these previous approaches are insufficient.

### 1.1 Genetic Programming

We use the five components of a genetic algorithm given in (Davis, 1987) (representation, evaluation function, initialization procedure, genetic operators, and parameters) as a framework for our discussion of



```
if  x > 0 then
   x := (x + 3) * (4 - y);
else
   y := x + y - 1;
end if;
```

Figure 1: A subroutine and an equivalent parse tree.

the 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 1 is represented by the parse tree shown in Figure 1. 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 1 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 (1992) 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.]

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 (1992) 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 no matter 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 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 (1992) 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-

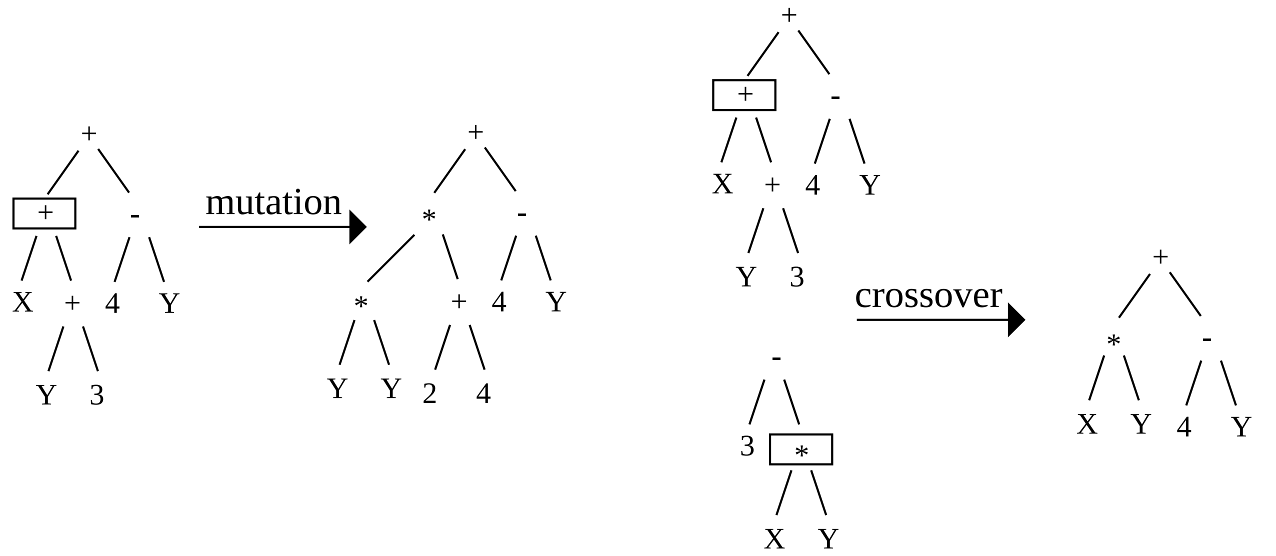


Figure 2: Mutation and crossover for genetic programming.

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 2.

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 2.

(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.

## 1.2 Type Constraints in Genetic Programming

Data structures are a key concept in computer programming. They provide a mechanism to group together data which logically belong together and to manipulate this data as a logical unit. Data typing (which is implemented in many computer languages including C++, Ada, Pascal and LISP) is a way to associate a type (or class) with each data structure instance to allow the code to handle each data structure differently based on its type. Even data structures with the same underlying physical structure can have different logical types; for example, a 2x3 matrix, a 6-vector, and an array of 3 complex number will likely have the same physical representation but will have different data types. Different programming languages use data typing differently. Strongly typed languages, such as Ada and Pascal, use data types at program generation time to ensure that functions only receive as arguments the particular data types they are expecting. Dynamically typed languages, such as LISP, use data types at program execution time to allow programs to handle data differently based on their types.

Standard genetic programming is not designed to handle a mixture of data types. In fact, one of its assumptions is "closure", which states that any non-terminal should be able to handle as an argument any data type and value returned from a terminal or non-terminal. While closure does not prohibit the use of multiple data types, forcing a problem which uses multiple data types to fit the closure constraint can severely and unnecessarily hurt the performance of genetic programming on that problem. We now discuss some cases where problems involving multiple data types can be adapted to fit the closure constraint naturally and without harm to performance. We then examine some problems where adapting to the closure constraint is unnatural and detrimental to performance and discuss some ways to extend genetic programming to eliminate the closure constraint.

One way to get around the closure constraint is to carefully define the terminals and non-terminals so as to not introduce multiple data types. For example, a Boolean data type (which can assume only two values, true and false) is distinct from a real-valued data type. Koza (1992) avoids introducing Boolean data types using a number of tricks in his definitions of functions. First, he avoids using predicates such as FOOD-HERE and LESS-THAN, which return Boolean values, and instead uses branching constructs such as IF-FOOD-HERE and IF-LESS-THAN, which return real values (or, more precisely, the same types as the arguments which get evaluated). Second, like the C and C++ programming languages, Koza uses functions which treat real-valued data as Boolean, considering a subset of the reals to be "true" and its complement to be "false". For example, IFLTZ is an IF-THEN-ELSE construct which evaluates and returns its second argument if the first argument is less than zero and evaluates and returns its third argument otherwise. Note that Koza's partitioning of the reals is more likely to yield true branching (as opposed to consistent execution of a single subtree) than the C language practice of considering 0 to be false and everything else true.

However, this approach is limited in its applicability. For example, in the matrix/vector manipulation problems and list manipulation problems described in Section 3, there is no way to avoid introducing multiple data types. (For the former, we need both vectors and matrices, while for the latter we need both lists and list elements.)

A way to simultaneously enforce closure and allow multiple data types is through the use of dynamic typing. In this approach, each non-terminal must handle as any of its arguments any data type that can be returned from any terminal or non-terminal. There are two (not necessarily exclusive) ways to do this. The first is to have the functions actually perform different actions with different argument types. The second is to have the functions signal an error when the arguments are of inconsistent type and assign an infinitely bad evaluation to this parse tree.

The first approach, that of handling all data types, works reasonably well when there are natural ways to cast any of the produced data types to any other. For example, consider using two data types, REAL and COMPLEX. When arithmetic functions, such as +, -, and *, have one real argument and one complex argument, they can cast the real number to a complex number whose real part is the original number and whose complex part is zero. Comparison operators, such as IFLTZ, can cast complex numbers to reals before performing the comparison either by using the real portion or by using the magnitude.

However, often there are not natural ways to cast from one data type to another. For example, consider trying to add a 3-vector and a 4x2 matrix. We could consider the matrix to be an 8-vector, throw away its last five entries, and then add. (In fact, that is exactly what we do in an experiment described in Section 3.2.) The problem with such unnatural operations is that, while they may succeed in finding a solution for a particular set of data, they are unlikely to be part of a symbolic expression that can generalize to new data (a problem which is demonstrated in our experiment). Therefore, it is usually best to avoid such "unnatural" operations and restrict the operations to ones which make sense with respect to data types.

[This bias against unnatural operations is an example of what in machine learning is called "inductive bias". Inductive bias is the propensity to select one solution over another based on criteria which reflect experience with similar problems. For example, in standard genetic programming, the human's choice of terminal and non-terminal sets and maximum tree size, provides a definite inductive bias for a particular problem. Ensuring consistency of data types, mechanisms for which we discuss below, is an inductive bias which can be enforced without human intervention.]

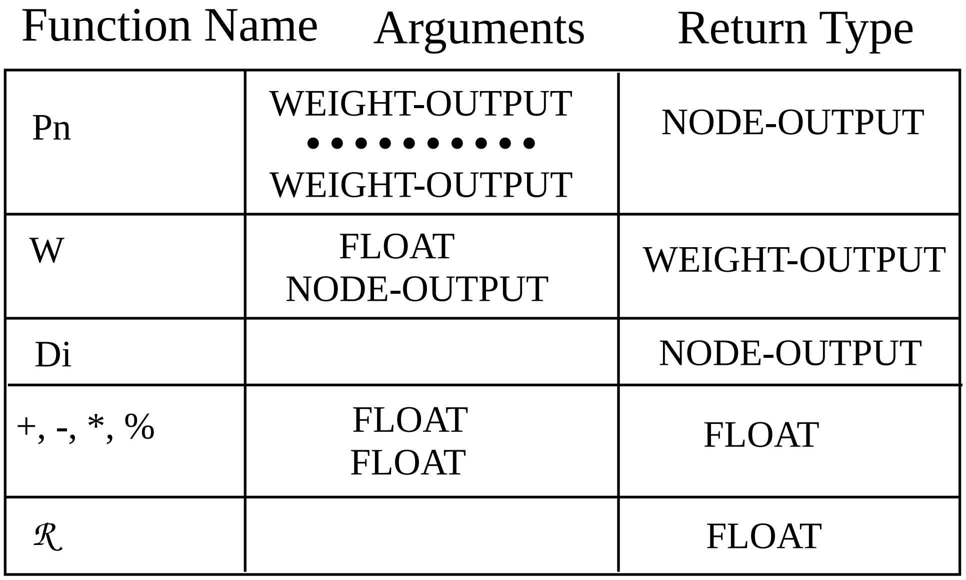
There is a way to enforce data type constraints while using dynamic data typing, which is to return an infinitely bad evaluation for any tree in which data type constraints are violated. The problem with this approach is that it can be terribly inefficient, spending most of its time evaluating trees which turn out to be illegal (as we demonstrate in an experiment described in Section 3.2).

[Note: Perkis has developed a version of his stack-based genetic programming (Perkis, 1994) which handles data type constraints using dynamic data typing. He defines multiple stacks, one for each data type, and functions which take their arguments from the appropriate stack. Due to the fact that stack-based genetic programming is new and unproven in comparison with Koza's tree-based genetic programming, we do not investigate its comparative merits here but do mention it as a potential future approach.]

A better way to enforce data type constraints is to use strong typing and hence to only generate parse trees which satisfy these constraints. This is essentially what Koza (1992, chapter 19) does with "constrained syntactic structures". For problems requiring data typing, he defines a set of syntactic rules which state, for each non-terminal, which terminals and non-terminals can be its children nodes in a parse tree. He then enforces these syntactic constraints by applying them while generating new trees and while performing genetic operations. Koza's approach to constrained syntactic structures is equivalent to our basic STGP (i.e., STGP without generic functions and generic data types), described in Section 2.1, with the following difference. Koza defines syntax by directly specifying which children each non-terminal can have, while STGP does this indirectly by specifying the data types of each argument of each non-terminal and the data types returned by each terminal and non-terminal.

To illustrate this difference, we consider the neural network training problem discussed by Koza. The terminal and non-terminal sets are $\mathcal{T}$ = {D0, D1, $\mathcal{R}$} and $\mathcal{N}$ = {P2, P3, P4, W, +, -, *, %}, where the Di are the input data, $\mathcal{R}$ is a floating-point random constant, the Pi are processing elements which sum their $n$ inputs and compare them with a threshold (assumed to be 1.0) to compute their output, W is a weighting function which multiplies its first input by its second input, and the rest are arithmetic operations. Koza gives five rules for specifying syntax

- The root of the tree must be a Pn.
- All the children of a Pn must be W's.



| Function Name | Arguments | Return Type |
|---|---|---|
| Pn | WEIGHT-OUTPUT<br>● ● ● ● ● ● ● ● ● ●<br>WEIGHT-OUTPUT | NODE-OUTPUT |
| W | FLOAT<br>NODE-OUTPUT | WEIGHT-OUTPUT |
| Di | | NODE-OUTPUT |
| +, -, *, % | FLOAT<br>FLOAT | FLOAT |
| $\mathcal{R}$ | | FLOAT |

Figure 3: Data types for Koza's neural net problem.

- The left child of a W must be $\mathcal{R}$ or an aritmetic function (+, -, *, or %).
- The right child of a W must be a Di or a Pn.
- Each child of an arithmetic function must be either $\mathcal{R}$ or an arithmetic function.

To instead specify this syntax using STGP, we would introduce three data types: NODE-OUTPUT, WEIGHT-OUTPUT, and FLOAT. Then, Figure 3 shows the data types associated with each terminal and non-terminal. Note that these data type specifications along with the constraints that a parse tree should return type NODE-OUTPUT and that a tree be at least depth two are equivalent to Koza's syntactic specifications. The advantage of the STGP approach over Koza's is that STGP does not require knowledge of what functions are in the terminal and non-terminal sets in order to define the specifications for a particular function. This is not a real advantage for a problem such as this where the functions are problem-specific. However, for functions such as VECTOR-ADD-3, which can be used in a wide variety of problems, it is much more convenient and practical to just say that it takes two arguments of type VECTOR-3 and produces a VECTOR-3 instead of trying to figure out all the other functions which could be its children for each problem.

However, this difference between Koza's constrained syntactic structures and simple STGP is relatively minor. The big new contribution of STGP is the introduction of generic functions (Section 2.2) and generic data types (Section 2.3). We have two factors motivating our introduction of this extra power (and bookkeeping complexity). The first, and more immediate, motivation is to solve problems involving symbolic manipulation of vectors and matrices. Such manipulation is a central part of many problems in estimation, prediction and control of multi-dimensional systems. These areas have been important proving grounds for other modern heurstic techniques, particularly neural networks and fuzzy logic, and could also be important applications for genetic programming. (In Section 3.3 we discuss a partial discovery of one of the classic examples in the estimation field, the Kalman filter.)

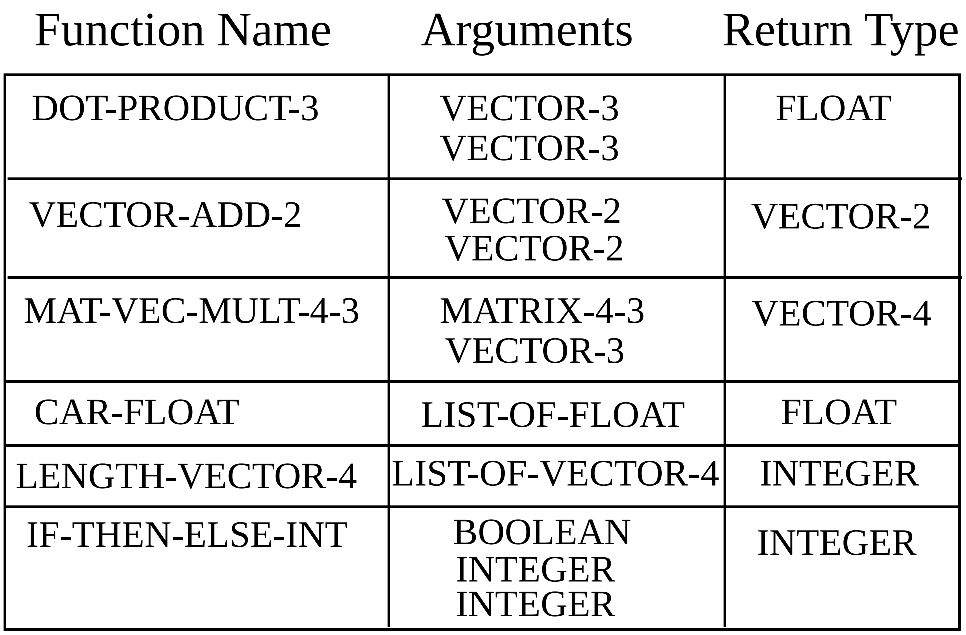
Some of the type constraints involving vectors and matrices are simple ones like those for the neural network problem above. For example, one should not add a vector and a matrix. To avoid this type mismatch, we can define two functions, VECTOR-ADD and MATRIX-ADD, the former taking two vectors as arguments and the latter taking two matrices as arguments. However, some of the type constraints are more complicated ones involving dimensionality. For example, it is inappropriate to add a 2x4 matrix and a 3x3 matrix or a 4-vector and a 2-vector. To handle these dimensionality constraints, we can define functions such as MATRIX-ADD-2-3, designed to handle structures of particular dimensions, in this case matrices of size 2x3. (We use such functions in our discussion of basic STGP in Section 2.1.) The problem is that there are generally a variety of dimensions. For example, in the Kalman filter problem there are three different vector dimensions (one for the state vector, one for the system dynamics noise vector, and one for the measurement noise vector) and $3^2 = 9$ different matrix shapes. Defining nine different MATRIX-ADD's is impractical, especially since these will have to change each time the dimensions of the data change. Using generic functions (described in detail in Section 2.2) allows us to define a single function MATRIX-ADD, which adds any two matrices of the same size. There is additional accounting overhead required to handle generic functions (describe in Section 2.2), but implementing this is a one-time expense well worth the effort in this case.

The second motivation for introducing generics is to take a (small) step towards making genetic programming capable of creating large and complex programs rather than just small and simple programs. As discussed by (Koza, 1994) and (Angeline & Pollack, 1993), the key to creating more complex programs is the ability to create intermediate building blocks capable of reuse in multiple contexts. Allowing these intermediate functions that are learned to be generic functions provides a much greater capability for reuse. The use of generic data types during the learning process makes it possible to learn generic functions.

In Sections 3.4 and 3.5 we provide two examples of learning building block functions, the list manipulation functions NTH and MAPCAR. As we discuss in Section 2, lists can be of many different types depending on their element type, e.g. LIST-OF-FLOAT, LIST-OF-VECTOR-3, and LIST-OF-LIST-OF-FLOAT. By using generic data types, we ensure that the NTH and MAPCAR we learn are generic, i.e. they can operate on any list type and not just on a particular one.

# 2 Strongly Typed Genetic Programming (STGP)

We now provide the details of our method of enforcing type constraints in genetic programming, called strongly type genetic programming (STGP). Section 2.1 discusses the extensions from basic genetic programming needed to ensure that all the data types are consistent. 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.



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

Figure 4: Some strongly typed functions with their 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 $\pi$ 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 4 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, 1984). 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 $i = 1, 2, 3, 4, ....$)

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

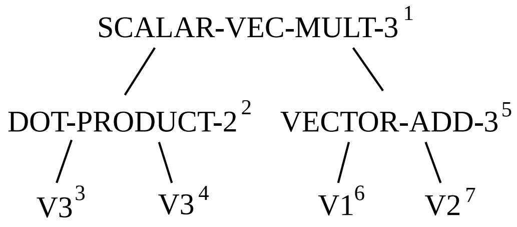


Figure 5: An example of a legal tree for return type VECTOR-3.

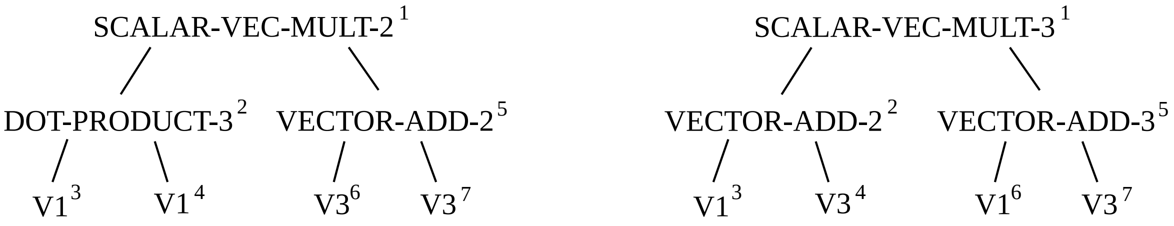


Figure 6: Two examples of illegal trees for return type VECTOR-3.

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 5 shows an example of a legal tree. Figure 6 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 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 5. 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, note that there is for the basic STGP described in this subsection (i.e., STGP without generics) the option of not enforcing this restriction. If we get to the maximum depth and cannot select a terminal of the appropriate type, then we can just throw away the tree and try again. This saves all the bookkeeping described below. However, once we introduce generics, looking ahead to see what types are possible is essential.

To implement this 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

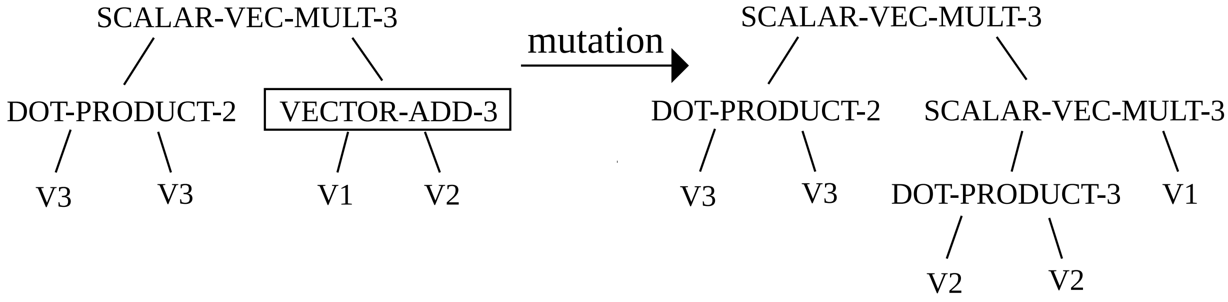


Figure 7: Mutation for STGP.

```
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 possibilities 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 7). 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 8). 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.

## 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

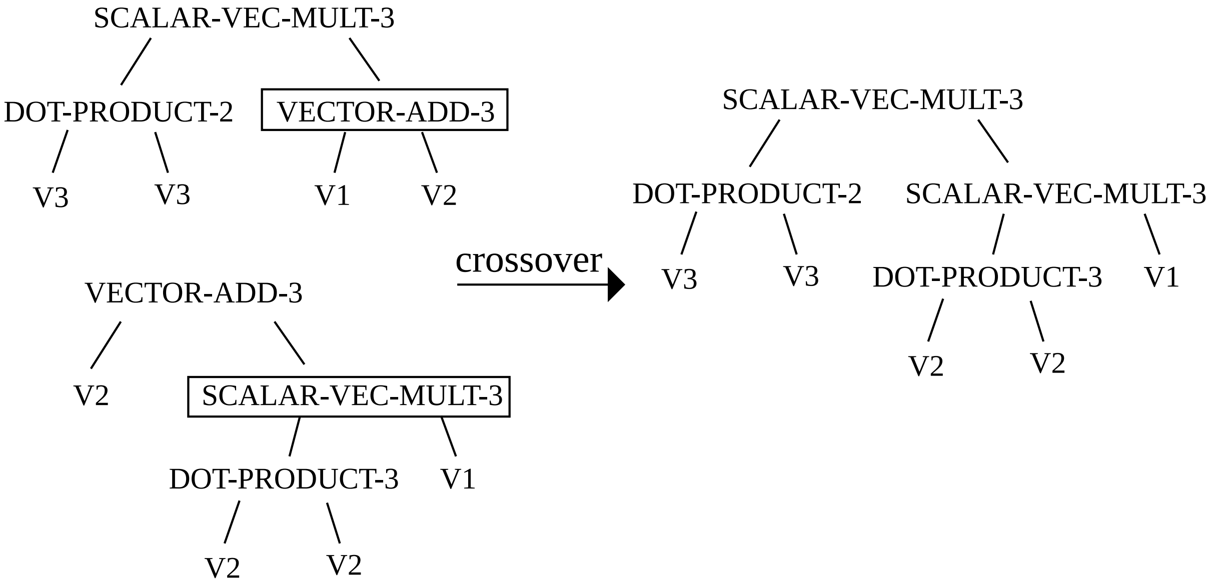


Figure 8: Crossover for STGP.

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.

[Note: The name "generic functions" as well as their use in STGP is based on generic functions in Ada (Barnes, 1982). Like generic functions in STGP, generic functions in Ada lessen the burden imposed by strong typing and increase the potential for code reuse.]

Some examples of generic functions are shown in Figure 9. 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 only to 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,

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

Figure 9: Some generic functions with their argument types and return types. Here, $i$ and $j$ are arbitrary integers and $t$ is an arbitrary data type.

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;
```

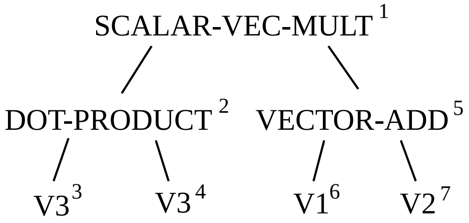


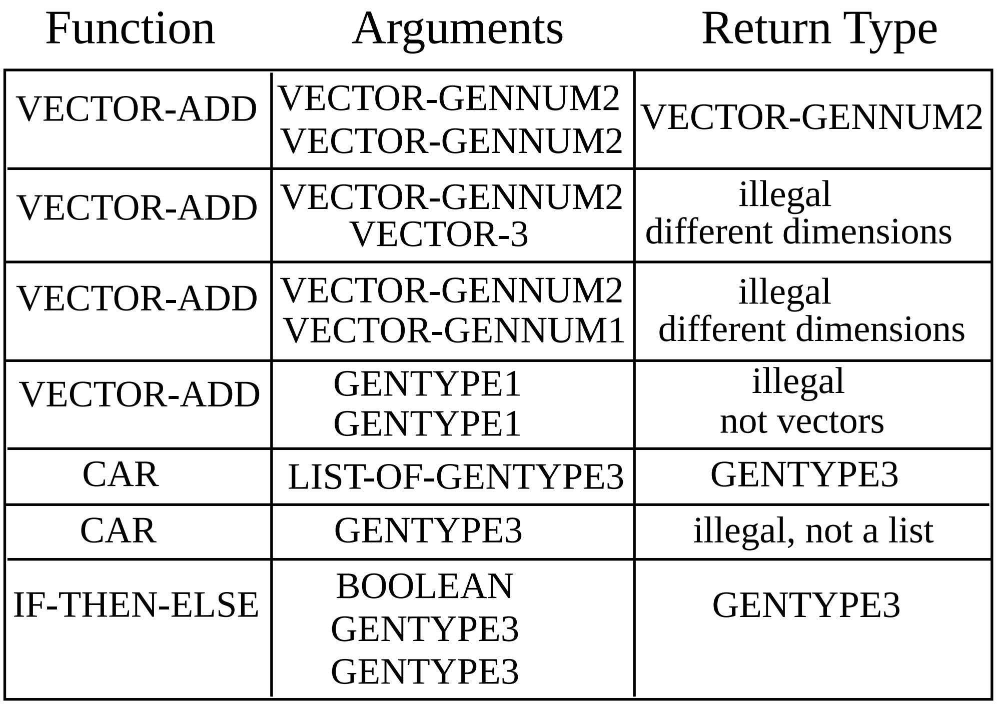
Figure 10: A legal tree using generic functions.

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, 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 10 shows the equivalent of the tree in Figure 5. To generate the tree shown in Figure 10 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 11.



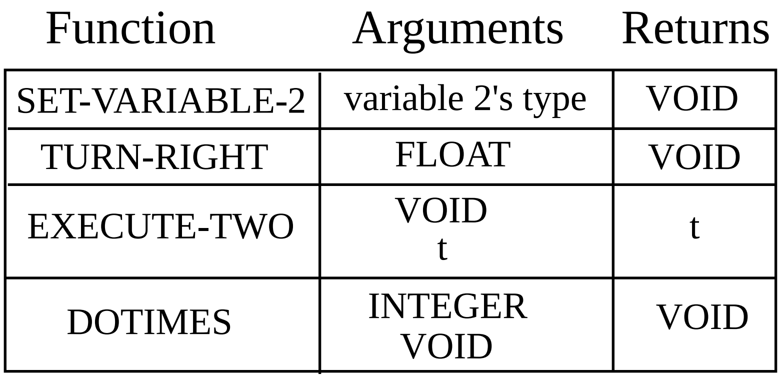
| Function | Arguments | Return Type |
|---|---|---|
| VECTOR-ADD | VECTOR-GENNUM2<br>VECTOR-GENNUM2 | VECTOR-GENNUM2 |
| VECTOR-ADD | VECTOR-GENNUM2<br>VECTOR-3 | illegal<br>different dimensions |
| VECTOR-ADD | VECTOR-GENNUM2<br>VECTOR-GENNUM1 | illegal<br>different dimensions |
| VECTOR-ADD | GENTYPE1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

Figure 11: Some examples of how generic data types are manipulated.

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 $\mathcal{T} = \{\text{V1, V2, V3}\}$ 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.

**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 10.



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

Figure 12: Some examples of functions which use the VOID data type.

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. Stated otherwise, generic data types provides an additional inductive bias against solutions which will not generalize well to data types which are not represented in the training data but which we nonetheless expect the solutions to handle.

Another advantage of using 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 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 & Ritchie, 1978). 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 12. 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 is another example of inductive bias, in this case towards simpler solutions.

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

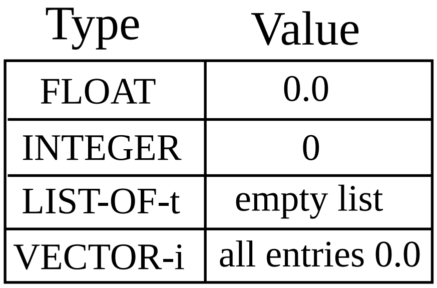
Most high-level programming languages provide local variables, which are slots where data can be stored during the execution of a subroutine. STGP also provides local variables. Like the terminal and non-terminal sets, the local variables and their types have to be specified by the user. For example,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

[Note that this means that the user has to have some insight into possible solutions in order to select the variables correctly. Of course, such insight is also required for selection of terminals and non-terminal sets. This is a shortcoming of genetic programming (strongly typed or not) which should be addressed in the future.]

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 13 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



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

Figure 13: Default values for different variable types.

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 and provide a mechanism for handling errors inside the tree.

Note that this approach to error handling is the opposite of the of Koza, who specifies that functions should return values under all circumstances. For example, his protected divide returns one when division by zero is attempted. We have two reasons for preferring our approach: (i) as discussed in Section 1.2 and demonstrated in Section 3.2, returning arbitrary values leads to poor generalization, and (ii) it is usually better to receive no answer than a false answer believed to be true. (Below, we give examples of how returning an arbitrary value can lead to false results.)

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 $\epsilon$, 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 $A^T A$ 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 $A(A^T A)^{-1}$ 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 ≥ 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, 1982). The essential ingredient it takes from Lisp is the concept of having programs basically be their parse trees (Steele, 1984). 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 type-consistent programs, which have the advantages discussed in Section 1.2 and demonstrated in Section 3.2. [Note: Some amount of strong typing has been added to Common Lisp with the introduction of the Common Lisp Object System (CLOS) (Keene, 1989). However, CLOS still uses dynamic typing in many cases.]

# 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, 1991). One important distinction of this genetic algorithm is its use of steady-state replacement (Syswerda, 1989) 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 reproduction rather than waiting to use them until the rest of the population has been evaluated, hence speeding up the progress of the genetic algorithm. In our implementation, we ensure that the newly generated individual is unique before evaluating it and installing it in the population, hence avoiding duplicate individuals. 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, 1991), such a comparison is fair.)

A second important features of this code is the use of exponential fitness normalization (Cox, Davis, & Qiu, 1991). 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, 1995). 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.

One aspect of our genetic algorithm which might strike some readers as unusual is the very large population sizes we use in certain problems. The use of very large populations is a technique that we have found in practice to be very effective for problems which are hard for genetic algorithms for either of the following reasons: (i) the minimal size building blocks are big (and hence require a big population in order to contain an adequate sampling of these building blocks in the initial population) or (ii) there is a very strongly attracting local optimum which makes it difficult to find the global optimum (and hence the population has to remain spread out over the space to avoid premature convergence). The second case also requires a very slow convergence rate, i.e. a parent scalar very close to one. When using large populations, steady-state reproduction is important; the ability to use a good individual as a parent immediately rather than waiting until the end of a generation matters more when the size of such a generation is large. Note that all the STGP examples converge within at most what would be three or four generations for a generational replacement algorithm. This does not mean that we are doing random search but rather that the steady-state approach is taking many "steps" in a single "generation".

In most cases, we ran the genetic algorithm with strong typing turned on in order to implement STGP as described in Section 2. However, for two experiments described in Section 3.2, we used dynamic typing (see Section 1.2) instead.

### 3.2 Multi-Dimensional Least Squares Regression

**Problem Description:** The multi-dimensional least squares regression problem can be stated as follows. For an $m$x$n$ 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 & 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:** $\mathcal{T} = \{\mathrm{A}, \mathrm{B}\}$

**Non-Terminal Set:** We use four different non-terminal sets for four different experiments

$$\begin{aligned}
\mathcal{N}_1 &= \{\text{MATRIX\_TRANSPOSE}, \text{MATRIX\_INVERSE}, \text{MAT\_VEC\_MULT}, \text{MAT\_MAT\_MULT}\} \quad (3)\\
\mathcal{N}_2 &= \{\text{MATRIX\_TRANSPOSE}, \text{MATRIX\_INVERSE}, \text{MAT\_VEC\_MULT}, \text{MAT\_MAT\_MULT},\\
&\quad \text{MATRIX\_ADD}, \text{MATRIX\_SUBTRACT}, \text{VECTOR\_ADD}, \text{VECTOR\_SUBTRACT},\\
&\quad \text{DOT\_PRODUCT}, \text{SCALAR\_VEC\_MULT}, \text{SCALAR\_MAT\_MULT}, +, -, *\} \quad (4)\\
\mathcal{N}_3 &= \{\text{DYNAMIC\_TRANSPOSE\_1}, \text{DYNAMIC\_INVERSE\_1}, \text{DYNAMIC\_MULTIPLY\_1},\\
&\quad \text{DYNAMIC\_ADD\_1}, \text{DYNAMIC\_SUBTRACT\_1}\} \quad (5)\\
\mathcal{N}_4 &= \{\text{DYNAMIC\_TRANSPOSE\_2}, \text{DYNAMIC\_INVERSE\_2}, \text{DYNAMIC\_MULTIPLY\_2},\\
&\quad \text{DYNAMIC\_ADD\_2}, \text{DYNAMIC\_SUBTRACT\_2}\} \quad (6)
\end{aligned}$$

The sets $\mathcal{N}_1$ and $\mathcal{N}_2$ are for use with STGP. $\mathcal{N}_1$ is the minimal non-terminal set necessary to solve the problem. $\mathcal{N}_2$ is a larger than necessary set designed to make the problem slightly more difficult (although as we will see it is still too easy a problem to challenge STGP).

The sets $\mathcal{N}_3$ and $\mathcal{N}_4$ are designed for use with dynamic data typing. The functions in $\mathcal{N}_3$ implement the same functions as $\mathcal{N}_2$ when the arguments are consistent and raise an error (causing an infinitely bad evaluation) otherwise. DYNAMIC-ADD-1 implements + if both arguments are of type FLOAT, VECTOR-ADD if both arguments are vectors of the same dimension, and MATRIX-ADD if both arguments are matrices of the same dimensions; otherwise, it raises an error. DYNAMIC-SUBTRACT-1 similarly covers -, VECTOR-SUBTRACT, and MATRIX-SUBTRACT. DYNAMIC-MULTIPLY-1 covers *, MAT-VEC-MULT, MAT-MAT-MULT, DOT-PRODUCT, SCALAR-VEC-MULT, and SCALAR-MAT-MULT; DYNAMIC-INVERSE-1 covers MATRIX-INVERSE; DYNAMIC-TRANSPOSE-1 covers MATRIX-TRANSPOSE.

The functions in $\mathcal{N}_4$ are the same as those in $\mathcal{N}_3$ except that instead of raising an error when the data types are inconsistent they do some transformation of the data to obtain a value to return. DYNAMIC-ADD-2 and DYNAMIC-SUBTRACT-2 will transform the second argument to be the same type as the first using the following rule. Scalars, vectors and matrices (using a row-major representation) are all considered to be arrays. If the second array is smaller than the first, it is zero-padded; if the second array is larger than the first, it is truncated. Then, the elements of the array are added or subtracted.

DYNAMIC-MULTIPLY-2 will switch the first and second arguments if the first is a vector and the second is a matrix. The dimensions of the second argument will be altered to match the dimensions of the first as follows. If this second argument is a vector, its dimension is increased by adding zeroes and decreased by truncating. If it is a matrix, its number of rows is increased by adding rows of zeroes and decreased by deleting rows at the bottom.

DYNAMIC-TRANSPOSE-2 does nothing to a scalar, turns an n-vector into a 1xn matrix, and does the obvious to a matrix. DYNAMIC-INVERSE-2 takes the reciprocal of a scalar (or if the scalar is zero, returns 1), takes the reciprocal of each element of a vector, takes the inverse of an invertible square matrix (or if the matrix is singular, returns the identity), and does nothing to a non-square matrix.

**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 MAX-DEPTH to be 12 in all cases. For the four experiments using $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, we used population sizes of 50, 2000, 50,000 and 10,000 respectively. (As explained below, the 50,000 value was to ensure that at least some of the members of the initial population were type-consistent trees.) We chose PARENT-SCALAR to be 0.99 and 0.998 for $\mathcal{N}_3$ and $\mathcal{N}_4$ respectively. (Because the experiments with $\mathcal{N}_1$ and $\mathcal{N}_2$ found optimal solutions in the initial population, the value of PARENT-SCALAR is irrelevant for them.)

**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 4.5 optimal trees in the initial population of 2000. Of the 45 total optimal trees generated over the ten runs, there were 30 distinct trees.

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 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 compare strong typing with dynamic typing, we ran the dynamically typed version of genetic programming ten times each for the non-terminal sets $\mathcal{N}_3$ and $\mathcal{N}_4$. For each run using $\mathcal{N}_3$, after 100,000 evaluations the best solution was always a tree which when evaluated returned a vector of three zeroes. This is significantly different from the best possible solution. The reason that this approach did not succeed was because by far the majority of the time was spent deciding that particular trees were not type consistent. For example, in an initial population of 50,000, only on the order of 20 trees were type consistent. Figure 14 shows why so few trees were type consistent; it tells the overall fraction of trees of different sizes which are type consistent, and it is clear that not very many are.

| Max Depth | Legal Trees | Total Trees | Fraction Legal |
|---|---|---|---|
| 1 | 0 | 2 | 0.0 |
| 2 | 0 | 18 | 0.0 |
| 3 | 3 | 1010 | 3.0e-3 |
| 4 | 195 | 3.1e6 | 6.3e-5 |
| 5 | 5.7e5 | 2.8e13 | 2.0e-8 |
| 6 | 5.8e12 | 2.4e27 | 2.4e-15 |

(a) Grow

| Max Depth | Legal Trees | Total Trees | Fraction Legal |
|---|---|---|---|
| 1 | 0 | 2 | 0.0 |
| 2 | 0 | 16 | 0.0 |
| 3 | 2 | 800 | 2.5e-3 |
| 4 | 76 | 1.9e6 | 4.0e-5 |
| 5 | 73280 | 1.1e13 | 6.7e-9 |
| 6 | 6.3e10 | 3.7e26 | 1.7e-16 |

(b) Full

Figure 14: Number of type-consistent trees vs. total number of trees for different tree sizes using the non-terminal set $\mathcal{N}_3$.

[Note: To understand the significance of Figure 14, one must remember the following: all trees which are type inconsistent will raise an error when executed because, for the particular non-terminal set used, all the non-terminals always execute all of their children. Hence, the column showing the fraction of legal trees is the factor by which STGP reduces the size of the search space without throwing away any good solutions.]

The ten runs with $\mathcal{N}_4$, each executed for 100,000 evaluations, did succeed in the sense that they did find trees returning 3-vectors very close to the optimum. Six of these runs were equal to the optimal to 6 places (which is all we recorded), one was the same as the optimal to 5 places, and the remaining three were the same as the optimal to 4 places. The problem with this approach, in addition to requiring two orders of magnitude more evaluations, is that the solutions do not generalize well to new data. To test generalization, we set $A$ to be a random 7x5 matrix and $B$ to be a random 7-vector. The ten best individuals from the ten runs with $\mathcal{N}_4$ all produced very different evaluations on this test data. The best was about 2.5 times the error produced by an optimal tree, while the worst was about 300 times the optimal. As we stated earlier, STGP found the optimal tree. We can conclude that the solutions found using $\mathcal{N}_4$ are specific to the particular $A$ and $B$ used for training and do not generalize to different values and dimensions for $A$ and $B$.

**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. Generating parse trees without regard for type constraints and then throwing away those that are not type consistent is a very inefficient approach for reasons that are made clear by Figure 14. Generating parse trees without regard for type constraints and then using “unnatural” operations where the types are inconsistent can succeed in finding solutions to a particular set of data. However, a standard genetic algorithm or hillclimbing algorithm could easily do the same thing, i.e. find the three-vector $X$ which optimizes the expression $(AX - B)^2$ for a particular $A$ and $B$. The advantage of genetic programming is its ability to find the symbolic expression which gives the optimal solution for any $A$ of any dimensions and any $B$. Allowing unnatural operations makes it unlikely that genetic programming will find this symbolic expression and hence destroys the power of genetic programming.

### 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, 1960). A standard formulation of a Kalman filter is the following. Assume that the system follows the stochastic equation

$$\dot{\vec{x}} = A\vec{x} + B\vec{n}_1 \tag{7}$$

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

$$\vec{y} = C\vec{x} + \vec{n}_2 \tag{8}$$

where $\vec{y}$ is a $k$-dimensional output (or measurement) vector, $C$ is a $k$x$n$ matrix, and $\vec{n}_2$ is a $k$-dimensional noise vector which is Gaussian distributed with mean 0 and covariance the $k$x$k$ 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

$$\begin{aligned} \dot{\hat{\vec{x}}} &= A\hat{\vec{x}} + PC^T R^{-1}(\vec{y} - C\hat{\vec{x}}) && (9) \\ \dot{P} &= AP + PA^T - PC^T R^{-1} CP + BQB^T && (10) \end{aligned}$$

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 9. In the "Analysis" portion, we discuss why we have focused on just the state estimate part (i.e., Equation 9) and what it would take to simultaneously learn the covariance part (i.e., Equation 10).

**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:** $\mathcal{T} = \{\mathrm{A, C, R, P, Y, X\_EST}\}$

**Non-Terminal Set:**

$$\begin{aligned} \mathcal{N} = \{&\mathrm{MAT\_MAT\_MULT, MATRIX\_INVERSE, MATRIX\_TRANSPOSE} \\ &\mathrm{MAT\_VEC\_MULT, VECTOR\_ADD, VECTOR\_SUBTRACT}\} \end{aligned} \tag{11}$$

**Evaluation Function:** Before running the genetic algorithm, we created a track using Equations 7 and 8 with the parameters chosen to be

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, C = \begin{bmatrix} 1 & 2 & -1 \\ 3 & -1 & 1 \end{bmatrix} \tag{12}$$

$$B = \begin{bmatrix} 1 & -1 \\ 2 & 1 \\ 0 & -3 \end{bmatrix}, Q = \begin{bmatrix} 2.5 & -0.25 \\ -0.25 & 1.25 \end{bmatrix}, R = \begin{bmatrix} 0.05 & 0 \\ 0 & 0.05 \end{bmatrix} \tag{13}$$

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

$$\vec{x} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, P = 0 \tag{14}$$

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 14 and with $P = 0$. For each of the 800 time steps, update $P$ and $\hat{\vec{x}}$ according to

$$\hat{\vec{x}}_{new} = \hat{\vec{x}}_{old} + \delta t * (\text{value returned by tree}), P_{new} = P_{old} + \delta t * (\text{right}_{\text{h}}\text{and}_{\text{s}}\text{ide of Equation 10}) \tag{15}$$

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 9 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 9. 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 9. Letting this run continue eventually yielded a “best” tree which implemented the solution

$$\dot{\hat{\vec{x}}} = A\hat{\vec{x}} + PC^T R^{-1}(\vec{y} - C\hat{\vec{x}}) - P^2\hat{\vec{x}} \tag{16}$$

and which had a score of 0.00357 as compared with a score of 0.00458 for trees which implemented Equation 9.

**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 9) and the covariance (Equation 10). 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, 1984) 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

$$\begin{aligned}
\mathcal{N}_1 &= \{\text{CAR}, \text{CDR}, \text{EXECUTE\_TWO}, \text{DOTIMES}, \text{SET\_VAR\_1}\} && (17)\\
\mathcal{N}_2 &= \{\text{CAR}, \text{CDR}, \text{EXECUTE\_TWO}, \text{DOTIMES}, \text{SET\_VAR\_1}, \text{ONE}, \text{PLUS}, \text{MINUS}\} && (18)\\
\mathcal{N}_3 &= \{\text{CAR}, \text{CDR}, \text{EXECUTE\_TWO}, \text{DOTIMES}, \text{SET\_VAR\_1}, \text{ONE}, \text{PLUS}, \text{MINUS},\\
&\quad\ \text{MIN}, \text{LENGTH}\} && (19)
\end{aligned}$$

**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.

[Note: This is a complicated evaluation function. It is necessarily so due to the multiple modes of failure and the variety of boundary conditions to check. As genetic programming attempts to solve larger and more complex programming problems, the evaluation functions should become even more complicated, and the definition of appropriate functions may become a major issue.]

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 $N > \text{length}(L)$, 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 $N > \text{length}(L)$ 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, 1992): 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: In fact, in Common Lisp (Steele, 1984) MAPCAR is defined to take a variable number of lists as arguments. Due to our use of strong typing, we cannot implement this exact version of MAPCAR. However, we could implement MAPCAR2, MAPCAR3, etc. which take 2, 3, etc. lists as arguments. These lists can be of different types; the only restriction is that the function's argument data types match the data types of the corresponding list elements. Due to computational constraints, we have chosen here to learn the simplest case, that involving a single list.]

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:** $\mathcal{T} = \{\mathrm{L}, \mathrm{GET\_VAR\_1}, \mathrm{GET\_VAR\_2}\}$

**Non-Terminal Set:**

$$\begin{aligned}\mathcal{N} \;=\; & \{\mathrm{CAR}, \mathrm{CDR}, \mathrm{EXECUTE\_TWO}, \mathrm{DOTIMES}, \mathrm{SET\_VAR\_1}, \mathrm{SET\_VAR\_2}, \\ & \mathrm{LENGTH}, \mathrm{APPEND}, \mathrm{FUNARG}\} \end{aligned} \tag{20}$$

**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 use of generic data types allows us to use the identity function and still know that it is being applied exactly once.) The score $S_L$ for each list L given that executing the parse tree either produces an error or the list $L_r$ is

$$S_L = \begin{cases} -10 - 2 * \mathrm{length}(L) & \text{if error} \\ -2 * |\mathrm{length}(L) - \mathrm{length}(L_r)| + \sum_{e \in L} 2^{-\mathrm{dist}(e, L_r)} & \text{otherwise} \end{cases} \tag{21}$$

where $\mathrm{dist}(e, L_r)$ is $\infty$ 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 ensures that all parse trees which are generated obey any constraints on data types. By doing this, STGP can greatly decrease the search time and/or greatly improve the generalization performance of the solutions it finds. 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, more powerful, and more oriented towards code reuse.

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

Angeline, P. J., & Pollack, J. B. (1993). Coevolving High-level Representations. In C. G. Langton (Ed.), *Artificial Life III.* Reading, MA: Addison-Wesley.

Barnes, J. (1982). *Programming in Ada.* Reading, MA: Addison-Wesley.

Campbell, S. L. & Meyer, Jr., C. D. (1979). *Generalised Inverses of Linear Transformations.* London: Pittman.

Cox, Jr., A. L. Davis, L. & Qiu, Y. (1991). Dynamic Anticipatory Routing in Circuit-Switched Telecommunications Networks. In (Davis, 1991) (pp. 124–143).

Cramer, N. L. (1985). A Representation for the Adaptive Generation of Simple Sequential Programs. In J. J. Grefenstette (Ed.), *Proceedings of the First International Conference on Genetic Algorithms* (pp. 183–187). Hillsdale, NJ: Lawrence Erlbaum Associates.

Davis, L. (1987). *Genetic Algorithms and Simulated Annealing.* London: Pittman.

Davis, L. (1991). *Handbook of Genetic Algorithms.* Von Nostrand Reinhold.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading, MA: Addison-Wesley.

Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems. *Transactions of the ASME: Journal of Basic Engineering,* 82, 35–45.

Keene, S. (1989). *Object-Oriented Programming in Common Lisp.* Reading, MA: Addison-Wesley.

Kernighan, B. & Ritchie, D. (1978). *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall.

Koza, J. R. (1992). *Genetic Programming.* Cambridge, MA: MIT Press/Bradford Books.

Koza, J. R. (1994). *Genetic Programming II.* Cambridge, MA: MIT Press/Bradford Books.

Montana, D. (1991). Automated Parameter Tuning for Interpretation of Synthetic Images. In (Davis, 1991) (pp. 282–311).

Montana, D. (1995). Genetic Search of a Generalized Hough Transform Space. In preparation.

Perkis, T. (1994). Stack-Based Genetic Programming. In *Proceedings of the IEEE Conference on Evolutionary Computation.*

Steele, G. (1984). *Common Lisp.* Burlington, MA: Digital Press.

Syswerda, G. (1989). Uniform Crossover in Genetic Algorithms. In D. Schaffer (Ed.), *Proceedings of the Third International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann.