Mike J. Keith and Martin C. Martin

The purpose of our current research is to investigate the design and implementation of a Genetic Programming platform in C++, with primary focus on efficiency and flexibility. In this chapter we consider the lower level implementation aspects of such a platform, specifically, the Genome Interpreter. The fact that Genetic Programming is a computationally expensive task means that the overall efficiency of the platform in both memory and time is crucial. In particular, the node representation is the key part of the implementation in which the overhead will be magnified. We first compare a number of ways of storing the topology of the tree. The most efficient representation overall is one in which the program tree is a linear array of nodes in prefix order as opposed to a pointer based tree structure. We consider trade-offs with other linear representations, namely postfix and arbitrary positioning of functions and their arguments. We then consider how to represent which function or terminal each node represents, and demonstrate a very efficient one to two byte representation. Finally, we integrate these approaches and offer a prefix/jump-table (PJT) approach which results in a very small overhead per node in both time and space compared to the other approaches we investigated. In addition to being efficient, our interpreter is also very flexible. Finally, we discuss approaches for handling flow control, encapsulation, recursion, and simulated parallel programming.

13.1 Introduction

In this chapter we explore the lower level implementation issues surrounding what we call the *Genome Interpreter.* Provided is example code from 5 test programs which were used to evaluate performance. Section 13.8 summarizes the results of these tests and discusses the trade-offs involved with the various implementations.

For the upcoming discussion, what we call an interpreter specifies the following lower level aspects of the design:

- the raw node representation
- how a tree of nodes is represented
- the method for evaluating an individual node
- the method for evaluating the tree as a whole
- the methods for (or methods to assist) those genetic operators which are dependent on the node or tree representation.

A key point is that the interpreter specifies the node implementation which is the particular part of the platform-coding in which the overhead will be magnified. Therefore, the interpreter is the most crucial component in the overall design with respect to space/time efficiency.

In order to illustrate the extremity of the node magnification, consider an application which uses a population size M of 2000 programs in which the average size P_{ave} of each individual program tree consists of 200 nodes. Also, consider a typical scenario that, in order to establish the fitness of each program, it must be executed over 20 test cases (let C be the number of test cases required). Therefore, for each generation, the total number of nodes which must be processed N_p and stored N_s is:

$$\begin{aligned} N_p &= MP_{\text{ave}}C = 8,000,000 \\ N_s &= MP_{\text{ave}} = 400,000 \end{aligned} \tag{13.1}$$

For tougher problems, the magnification factor increases at least quadratically since both the program size and the population size must increase.

13.2 Pointer Based Implementations

Koza [1992] and Tackett [1993] offer pointer based implementations for use in genetic programming in which each program is a parse tree and each node contains a pointer to each child or input. This “traditional” approach for representing the tree structure is typically coded in C as shown in Figure 13.1.

Here, `RETURN_TYPE` stands for the data type that the node evaluates to. Note that the `Args` parameter is an array of pointers to other `Node`s and allows a collection of such `Node` structures to be connected into a tree. A memory management mechanism is required to allocated and de-allocated memory for these nodes on demand. One can typically execute such a tree structure as shown in Figure 13.2.

```
struct Node {
   unsigned char Type ;
   unsigned char Arity ;
   RETURN_TYPE Value ;
   struct Node *Args[MAX_ARGS] ;
   RETURN_TYPE ArgValues[MAX_ARGS] ;
   RETURN_TYPE (*EvalFunc)();  /* pointer to function that
         evaluates the subree rooted at this node. */
}
```

Figure 13.1
A traditional pointer-based tree structure coded in C.

```
RETURN_TYPE Eval (struct Node *N)
{
   if (N->Type == TERMINAL) return N->Value ;
   if (N->Type == MACRO) return (*N->EvalFunc)(N);
   else {
      for (i=0; i<N->Arity; i++)
         ArgValues[i] = Eval(N->Args[i]);
      return (*N->EvalFunct)(N);
   }
}
```

Figure 13.2
This routine recursively evaluates a subrtee; in the results section it is referred to as the “if statement tree”. If the subtree is a macro, we simply call the evaluation function. Otherwise, we recursively call `Eval()` on each

```
class Node { public: virtual RETURN_TYPE Eval()=0; };

class AddNode : public Node
{
   Node *Arg[2];
public:
   RETURN_TYPE Eval() {return Arg[0]->Eval() + Arg[1]->Eval();}
};

class VarNode : public Node
{
   int Index;
public:
   RETURN_TYPE Eval() {return VarTable[Index];}
};
```

Figure 13.3
An example of a pointer based tree representation in C++; in the results section it is referred to as the “virtual function tree”. Inheritance and virtual functions are used to ensure that the appropriate evaluation function is called and so that each node takes up only the amount of memory necessary.

This recursive routine will execute the genome in a post-order fashion (evaluating children before parent) unless the **`Type`** of the node is **`MACRO`**. In this case, the node must call **`Eval()`** to evaluate whichever of it’s own functions it needs.

The overhead in both time and memory for the example code above is significant. In many cases, the interpreter spends more time parsing the **`Type`** information than it takes to execute the node itself. In fact, because evaluating a typical GP token (like **`ADD`**, **`OR`**, **`const`**, etc.) is usually accomplished in a single machine instruction, it is imperative that we omit the type parsing completely. We can do this in C by making all functions macros and removing the **`Type`** field, or more elegantly in C++ using virtual functions as in Figure 13.3.

Obviously, this is a superior approach and reduces the node time-overhead. Note that the tree structure is executed in a pre-order traversal. A minor disadvantage to this coding is that the **`Eval`** method in each of the derived classes is slowed down by the virtual function mechanism used in C++ [Eckel 1990]. Therefore, we coded this approach without the inheritance in C which resulted in the fastest implementation in this chapter. The only overhead required to evaluate the child of a node is two pointer indirections, an array reference and a function call.

The minimum average memory used to represent a node in a pointer based representation can be calculated using a small trick. Each node, although it may have any number of children, has exactly one parent (except for the root). By associating each edge with it's child instead of it's parent, we see that in a tree of N nodes, there are $N-1$ edges. For large N, then, there is on average one edge per node and hence one **`Arg`** pointer per **`Node`**. The linear schemes below show how to represent a tree with no memory overhead at all used to record the topology of the tree. Overall, then, the efficiency of pointer-based representations is quite poor in comparison to the linearized implementations discussed below. This C++ approach above allows each class derived from **`Node`** to uniquely define its parameters thus minimizing the memory overhead and achieving the lower bound of one pointer per node to store structure. Therefore, this is our preferred pointer-based approach. We should also point out that all of the pointer-based implementations require a memory management service which adds some additional speed overhead and coding complexity to their implementations.

13.3 A Postfix, Stack-Based Approach

We now introduce a stack based approach in which each function and terminal is responsible for getting its own arguments (if any) by popping them from a stack and pushing its single output on a stack. This is similar to the FORTH programming Language (see Winfield [1989]). Consider the 7-node program below:

```
Postfix: a b + c d - +                  Infix: (a+b) + (c-d)
```

The stack convention just described provides an "implicit connection" between nodes. For example, terminal node **`a`** will push its value onto the empty stack, then **`b`** will do the same. At this point the stack contains the two values with **`b`**'s on top. The **`+`** node then pops two values (in this case **`a`** and **`b`**), adds them and puts the result back on the stack. This implicit connection allows us to omit the **`Arg`** pointers needed in the pointer based implementation and store the entire program as an array of nodes, as shown in Figure 13.4

```
class Program
{
   int Size;
   Node Genome[MAX_GENOME_SIZE];
   Stack stack;
public:
   RETURN_TYPE Eval() ;
};

RETURN_TYPE Program::Eval () {
   for (int i=0; i<Size; i++) { // For each node in order:
      switch (Genome[i].GetType()) {
      case ADD: { stack.PUSH(stack.POP() + stack.POP()) ; break; }
      case MULT: { stack.PUSH(stack.POP() * stack.POP()) ; break; }
      case VAR: { stack.PUSH(genome[i].GetVar()); break; }
      }
   }
}
```

Figure 13.4
The postfix program representation and an example tree evaluation function. There is no memory used to represent the tree topology.

Each program entity can be implemented as a record in which the program's tree is stored, not with pointers, but with a simple linear array. To execute such a program, we can simply index through the array and execute each node using a **`case`** statement as shown in the Figure.

Thus the node execution overhead in this implementation consists of 1 **`case`** lookup per node as well as 1 or 2 increments/decrements per node to maintain the stack, and an increment to move along the genome. Note that the **`GetType`** and **`GetVar`** routines are methods encapsulated in an unspecified **`Node`** class. A potential advantage of this scheme is that the base functions can be in-line thus avoiding the function call overhead. However, we found the **`case`** statement overhead in addition to the stack overhead made this approach relatively slow.

Stack-based implementations in general do not require the linear string of opcodes to represent a syntactically correct parse tree. This is a potential advantage of the stack-based approach since the other interpreters we considered require the syntax to be maintained.

We can see that our first linear approach meets many of the goals which we set out to address. We can directly access any entry in the genome-array which allows many manipulations of the genome to be simplified, especially choosing a random node. Furthermore, the array representation also allows the memory management mechanism to be completely omitted. The primary advantage of this approach, however, is that it provides large savings on the node size overhead since the implicit-connection feature uses no memory to represent the tree's topology.

13.3.1 Memory Efficiency

Using fixed-sized arrays to hold variable-sized Genomes can obviously result in a certain amount of space being unused. Therefore, although each node in an implicit-connection scheme can only take up 2 bytes, the effective node size (S_e) is greater than the actual node size (S_a) in relation to the average unused array space (U_{ave}) and average genome size (G_{ave}):

$$S_e = S_a + \frac{U_{\text{ave}}}{G_{\text{ave}}} \tag{13.2}$$

However, the fact that individuals tend to always use up as much space as they can, tends to minimize this effect and cause the array size to implicitly provide parsimony.

There is a way to further minimize this overhead and to allow the population to explore a variety solutions which have different space requirements on the genome. The idea is to simply provide a distribution of array sizes in the population. Each program structure has a **`TraitIndex`** parameter which specifies a set of characteristics for that individual or perhaps all individuals in a given region:

```
class Program { int Size; char TraitIndex; /* others...*/};
class Trait { int MaxSize, Minsize, MaxLoopDepth, /* others...*/};
```

Thus the **`TraitIndex`** is an index into an array of **`Trait`**s which allows us to reference the array size (**`MaxSize`**) of the individual. So if a given problem requires an expected genome size of N, we create a set of Individuals whose array sizes are distributed between *N-d* and *N+d* where *d* is the configurable array-size deviation. Note that the genetic operators must not perform an alteration on an individual which violates its **`MaxSize`** constraint. Also note that this still does not require us to use a memory manager since our variable-sized arrays remain static once created. In general, this trait-table approach allows us to explore a range of possibilities for any characteristic we chose.

13.3.2 Manipulating Postfix Programs

In order to faithfully implement the traditional GP operators, we must ensure that after initialization, crossover and mutation we have a valid representation of a tree. The basis of the following algorithms is to use the arity of every node to allow the syntax to be adhered to. At this point it should be mentioned that since none of these operations are performed while calculating fitness, their efficiency in general is not of prime concern.

13.3.2.1 Postfix Initialization

We can initialize a linear array of nodes as an implied tree in the same recursive manner that pointer based trees are initialized [Koza 1992]. One difference is that in addition to a `MaxDepth` constraint, one needs to also constrain the size of the implied tree to be no greater than the genome array size. This can be accomplished by having a parameter which keeps a count of the number of open branches and using it as follows:

```
if ((NumOpenBranches + CurrentSize) == TargetSize) Resolve = TRUE;
```

Where `Resolve` is a flag which forces all subsequent nodes to be terminals thus forcing all open branches to be resolved. A second difference associated with initializing a postfix tree is that we need to initialize from right to left so that we can begin with function nodes and end with terminal nodes. We conclude the initialization process by copying all of the nodes left so they start at the zero position in the array.

Note, however, that the above method will tend to produce lopsided trees. To avoid this, we can allocate enough memory to hold the biggest tree for the depth we want, and then recursively initialize the array, as show in Figure 13.5

However, the fact that we have a linearized implementation should motivate us to explore alternative techniques for various genetic operators. Based on the current scheme, one can initialize a valid postfix expression by simply requiring that the ongoing `StackCount`, as we scan from left to right, is never negative and that the final count is 1. Note that the `StackCount` for a token equals the number of arguments it places on the stack minus the number of arguments it takes off the stack. If we initialize in the reverse direction (from right to left as shown below) the cumulative `StackCount` never becomes positive until we reach the end at which point the overall sum still needs to be 1. One can also

```
int Program::Initialize (int depth, int max_args, int loc)
{
   if (depth == 0) Genome[loc].Initialize(0) ; // init to terminal
   else {
      int num_args = /* random number from 0 to max_args */ ;
      for (int i=0; i<num_args; i++) { // make the arguments
         loc = 1 + Initialize (depth-1, max_args, loc) ;
      }
      Genome[loc].Initialize(num_args) ;
   }
}
```

Figure 13.5
This method recursively initializes an instance of a `Program` class. It returns the last location in the array that it initialized, and inputs the depth of the tree to create, the maximum number of arguments that a node can have, and the location in the array at which to start.

```
Initialize (int TargetSize, int MinCount)
{
   int i, CurrentSize = 0;
   Node *NodePtr = Genome + MaxGenomeSize;

   while (1) {
      if ((++CurrentSize + abs(StackCount)) == TargetSize) ||
          (StackCount == MinCount))
         StackCount += GetRandomTerminal(--NodePtr);
      else if (StackCount == 0)
         StackCount += GetRandomFuntion(--NodePtr);
      else
         StackCount += GetRandomToken(--NodePtr);
      if (CurrentSize == TargetSize) break;
   }

   for (i=0; i<TargetSize; i++) //left shift the genome
   *NodePtr = *NodePtr++ + (MaxGenomeSize-TargetSize);
}
```

Figure 13.6
A method to initialize a genome using the constraint that the cumulative stack count must never become positive. Note that it initializes the genome from right to left, then shifts it to start at the proper place.

add another constraint, **`MinCount`**, which allows us to loosely control the implied shape of the linearized tree (see Figure 13.6).

13.3.2.2 Postfix Crossover

Our golden rule, that the **`StackCount`** sums to 1 over any legal postfix expression, can be extended to subtrees also. That is, if one starts at any point X on a postfix expression and sums leftwise until one reaches point Y where the **`StackCount`** first goes to 1, then points X and Y will span the subtree whose root node is X. Note that terminals constitute a subtree of size one and are, on average, the most commonly exchanged subtrees.

Since it is a simple matter of determining where a subtree ends, implementing the traditional GP crossover as a subtree exchange operation is straightforward. But again, since we are exploring linear approaches, it seems suitable to consider the implementation of a GA style, 1-point crossover operation [Holland 1975], [Goldberg 1989]. The stack counting mechanism described in the previous section is used for such a routine. The basic rule is that any 2 loci on the 2 parent genomes can serve as crossover points as long as the ongoing **`StackCount`** just before those points is the same. This requirement forces the syntax to be maintained. For example, consider the 2 parent genomes below, with the ongoing **`StackCount`** shown in brackets:

```
P1 = { a[1], b[2], a[3], -[2], +[1], c[2], -[1] }
P2 = { d[1], e[2], +[1], b[2], +[1] }
```

We can exchange the segment {**`+, b, +`**} on P2 with the segment {**`a, -, +, c, -`**}, {**`+, c, -`**} or {**`-`**} on P1 since the ongoing **`StackCount`** (shown in the brackets) just before all of these segments are the same (2).

13.3.2.3 Postfix Mutation

Mutation can be performed using part of the initialization and crossover methods. We select a subtree to replace as we do in crossover, and then generate a subtree just like we do in initialization. Finally, we may need to shift part of the non mutating segment in order to make room for the mutating part.

13.3.3 The Flow Control Problem with Postfix

The major problem with the postfix ordering scheme is its inability to handle flow control. The obvious dilemma is that with postfix, the arguments to a function are always executed before the function itself making it impossible to avoid the execution of conditional sub-expressions which we want to skip.

13.4 Mixfix

There is a way to avoid executing parts of a tree while satisfying the requirement that functions get and return values through the stack. We simply allow certain functions to perform arbitrary computations *between* evaluating arguments, and use the results of these computations to decide whether to evaluate the next subtree or skip it. Consider a hybrid ordering we call *mixfix* in which the arguments for flow control constructs are interspersed with code for the actual function. For example, the following coding:

```
( if Y then X - (a+b) ) + c
```

results in the following mixfix expression:

```
Y IF X a b + - ENDIF c +
```

Here we introduced a special marker token “**`ENDIF`**” which indicates where the “**`IF`**” function should skip to when its conditional (**`Y`**) is false. The delimiters are also needed in order to maintain syntax and avoid an expression where the **`IF`** construct is left without an **`ENDIF`**:

```
X Y Z IF + - //the AritySum is OK, but no ENDIF
```

This scheme adds complexity to the genetic operators since finding or creating subtrees for both the **`IF`** node and the **`ENDIF`** node must be accomplished in a different manner than the other tokens. In addition, nested conditional structures will require a counting operation so that a given conditional can find the **`ENDIF`** which belongs to it. This will certainly add execution overhead to the system. One way to minimize this overhead is to omit the **`ENDIF`** and use a *Jump-Offset* as part of the **`IF`** construct:

```
X IF:5 X a b + - c + //the IF will skip 5 tokens if Y is FALSE
```

Now, the **`IF`** function simply adds its Jump Offset to the current token index in order skip its “then” subtree as opposed to searching for an end marker. This scheme minimizes the execution overhead associated with the delimiter approach above. However, it adds even more complexity to the genetic operators since for expressions containing nested conditional structures, the Jump Offsets of the outer flow constructs must be adjusted when any of their inner subtrees are altered.

For flow constructs with more than one argument, we must distribute the individual sub-constructs as follows:

```
Y IF:4 a b + ELSE:3 c d -
```

In this case, if **`Y`** is true, the **`IF`** function immediately returns allowing the execution to continue at the **`a`** node. When the **`ELSE`** sub-construct is invoked it simply skips the **`c d -`** sub-expression. When **`Y`** is false, the **`IF`** function skips to the node immediately following the **`ELSE`** token (in this example the **`c`** node).

So now we have added even more complexity to the syntax preserving operations since the entire distributed **`IF-THEN-ELSE`** construct must be dealt with as a whole for various operations. For example, the crossover method obviously could not exchange the (**`ELSE:3 c d -`**) fragment by itself leaving the **`IF`** without an **`ELSE`**.

Although the stack based approaches just discussed have obvious drawbacks, they do offer some distinct advantages over the other implementations discussed throughout this chapter. The primary advantage of the stack based approaches are that they allows us to easily evaluate programs in parallel manner by maintaining a different stack for each one. This is very useful, for example, in studying emergent behavior, where one wishes to simultaneously evaluate a number of agents interacting in the same simulation. A second advantage is that syntactically unconstrained expressions are possible thus allowing for experimentation along those lines. Finally, by having an explicit stack, we can control what happens when a stack overflow occurs which is not possible for implementations which execute recursively using a system stack.

13.5 Prefix Ordering

We realized that the best way to naturally solve the flow control problem while still taking advantage of the memory efficiency of the linear implementation, was to simply use a prefix ordering on the genotype array. Our prefix ordering scheme has 3 potential advantages over postfix:

1. arguments must be explicitly evaluated by their parent which allows control constructs to be implemented in a natural manner.

2. the coding required to skip a subtree is quite simple and does not require the special mechanisms discussed previously in the mixfix section (like using jump-offsets or bracket tokens).

3. an explicit stack mechanism is not needed which reduces coding complexity and performance overhead.

We execute through the linear chromosome array recursively as shown in Figure 13.7. For efficiency, a global variable keeps track of the current position within the chromosome array. To evaluate the next argument (which is done with a call to **`EvalNextArg()`**) we

```
Node *current_node ; // The current Node being evaluated.

class Individual {
   Node *Genome ;  // pointer to an array of nodes.
public:
   RETURN_TYPE Eval()
      { current_node = Genome; return EvalNextArg() ; }
}

inline EvalNextArg() { return(current_node++).Eval() ; }
inline SkipNextArg()
 { for(int count=0; count>-1; count+=(current_node++).ArityM1()); }

class Node {
   RETURN_TYPE (*EvalFnct)() ;
public:
   RETURN_TYPE Eval() { return (*EvalFnct)() ; }
}

int x = 1.234 ;
float add() { return EvalNextArg() + EvalNextArg(); }
float varX() { return x ; }
```

Figure 13.7
The prefix representation and evaluation code, with example code for the nodes. Note that each **`Node`** must defin a method called **`Eval()`** which uses **`EvalNextArg()`** to get it's arguments.

simply increment the global pointer and call the evaluation function of the node. If this evaluation function needs arguments, it can call **`EvalNextArg()`** to get them; each call gets a different argument.

Note that the distributed evaluation approach makes aborting an individual programs in midstream difficult. Although this can be accomplished using the standard C library functions **`setjump()`** and **`longjump()`**, the evaluation schemes used in the postfix and mixfix implementations allow a program to be suspended or aborted at any node.

Our results in Section 13.8 show that the prefix approach is the most efficient. Also, from inspection of the above test programs, it should be apparent that the PJT scheme provides the cleanest implementation. A key point is to note that **`EvalNextArg()`** includes the important side effect operation of incrementing the node pointer. This can lead to problems for logical operations like **`AND`** which will skip evaluating (and therefore skipping) it's second argument if the first argument is false. Therefore, a safe implementation of **`AND`** would be:

```
int AND()
{
   if (!EvalNextArg()) { SkipNextArg(); return FALSE; }
   else return EvalNextArg() ;
}
```

13.5.1 Initialization, Crossover and Mutation with Prefix

The prefix ordering scheme allows syntax to be maintained in the same fundamental way as postfix does. That is, by taking the arity of each node minus 1, and summing from left to right, the overall sum must equal -1 in order for the prefix expression to be valid.

A key point with respect to initialization is that with prefix, we add terminals at the end as we scan from left to right which allows the left-shift operation coded in the postfix initialization routine to be bypassed. Otherwise, the initialization, crossover, and mutation implementations used for prefix are analogous to the postfix routines discussed in Section 13.3.2

13.5.2 Handling Program Flow with Prefix

In this section we discuss various ways to handle program flow in the prefix scheme. The fundamental idea is to use the **`EvalNextArg()`** routine to evaluate arguments and **`SkipNextArg()`** to skip over an argument without evaluating it. These routines are shown in Figure 13.8. Note that, in practice, we would keep track of how many iterations of the **`while`** loop we have performed and abort when the number becomes too large. The pur-

```
RETURN_TYPE if_then_else()
{
   if (EvalNextArg()) {
      const RETURN_TYPE result = EvalNextArg();
      SkipNextArg() ;
      return result ;
   } else {
      SkipNextArg() ;
      return EvalNextArg() ;
   }
}

RETURN_TYPE while ()
{
   Node *Start = current_node;

   while (EvalNextArg()) {
      EvalNextArg() ;
      current_node = Start ;
   }
   SkipNextArg() ; // skip the body of the loop
}
```

Figure 13.8
Implementing `if-then-else` and `while` in a prefix implementation.

pose of the given code, however, is just to show how to perform iteration in the prefix implementation.

13.6 The Node Representation

Up to this point, we have been concerned only with how to represent the *topology* of the tree, and have simply used a function pointer to represent the information needed to evaluate a node. In this section, we present our preferred approach for representing a node.

13.6.1 General Data Support

The idea can be seen to follow from a simple observation. If there are 256 different types of node (functions and terminals combined, each constant counting as a different type of node), then we only need one byte to represent the node. This byte could be used as an index into an array of information about that type, including a function pointer, arity, name, etc. The function pointed to we call the *handler*, the array we call a *jump table* (since it’s primary purpose is hold the function pointer), and the entire entry for one type of node is called a *token*, represented by the `Token` class:

```
class Token { char *Name; char ArityMinus1; float (*Funct)(); /*...*/};
```

This strategy of simply providing memory chunks (tables) based only on the data-types needed for an application as opposed to requiring a special token for each and every variable, allows the data-declaration portion of a program to be more loosely defined up front and even provides a framework for allowing such declarations to evolve if needed. We call this property *general data support*, and argue that for non-toy problems, the evolution of the programs data-model in addition to its parse tree might be essential and that general data support, while not a solution to this problem, is a step in the right direction.

13.6.2 The Opcode Format

For an application using 2 floating-point variables and many pre-defined constants, where one byte will do, one might set up the jump-table as follows:

Functions[0-15], Variables[16-17], Constants[18-255]

Note that in this scheme, the pointers to the constant handler is repeated over it's range and examines the node value itself to determine which constant is needed. See Figure 13.9. A pointer to the constant handler repeatedly appears in the jump-table in entries 18 through 255. When called, it finds which constant (between 18 and 255) is needed by examining the current node and returns the proper entry from it's array.

Although 256 different types may do in some applications, many applications need more (e.g. when using random ephemeral constants or other run-time-generated terminals such as modules). From here on we will assume that there are more than 256 but less than 65,536 types of node, so that two bytes is both necessary and sufficient. We could simply have an array of size 65,536, but this is cumbersome. Firstly, this takes a lot of memory and filling all entries is cumbersome. Secondly, many of these entries will be similar (e.g. many will be distinct constants). To take advantage of this, we can use part of the two bytes to represent the general type of node (e.g. "+", "sin", "constant"), and the rest of it (if needed) to specify the exact node type (e.g. "+" and "sin" need no further explanation; "constant" needs to know the particular value). The portion for the general type we call the *function index*, and the portion for the specific type we call the *specific lookup table index*, or simply the *table index*. We have one function for each general type of node; the function is passed the two-byte opcode and can use the table index however it sees fit.

There are many possibilities for splitting the 16 bits available to us. For most applications, a maximum of 16 functions would suffice. Therefore we considered using 4 bits to represent the function index and the remaining 12 bits for the table indexing. Thus our `Node` structure could be coded using bit-fields:

```
#define RETURN_TYPE float

class Token JumpTable[256] ; //maps node number to info about that node.

class Node *current_node ;  // The node being evaluated.

/* Function handlers */
RETURN_TYPE Add() { return EvalNextArg() + EvalNextArg() ; }
RETURN_TYPE Mult() { return EvalNextArg() * EvalNextArg() ; }
// other functions

/* Variable handlers */
RETURN_TYPE x, y ;
RETURN_TYPE X() { return x ; }
RETURN_TYPE Y() { return y ; }

/* Constant handler */
RETURN_TYPE ConstTable[255-18+1] ;
RETURN_TYPE Const() { return ConstTable[*current_node-18] ; }

RETURN_TYPE EvalNextArg() { return (*JumpTable[current_node++].funct)() ; }
class Node genome[] = { 0/*add*/, 1/*mult*/, 16/*x*/, 17/*y*/, 16/*x*/ };

void InitJumpTable()
{
   JumpTable[0].name = "add" ; JumpTable[0].ArityM1 = 1 ;
   JumpTable[0].funct = Add ;
   /* other functs and vars*/

   for (int i=18; i<=255; i++)
      JumpTable[i].funct = Const, JumpTable[i].ArityMinus1 = -1 ;
}
```

Figure 13.9
An example of the workings of the one byte opcode. To complete the code we also need to initialize the name and value of each constant, and to initialize the other function entries in the jump table. Also, the genome shown would be for testing purposes; the real genomes would genomes would be generated at run time.

```
class Node {
   unsigned functIndex:4; //function index
   unsigned tableIndex:12; //table index
   // ...
} ;
```

However, with this setup, all references of the function index and table index require both a bitwise-AND operation and a bit-SHIFT operation which adds considerable node overhead to the interpreter. Therefore, there is really no choice but to place the 2 indexes on byte boundaries:

```
class Node { unsigned char functIndex, tableIndex; };
```

The only possible drawback to this layout is that now we have 256 possible tables (this is probably too many), each with room for 256 entries for variable/module/constant lookup (this is probably too few especially for constants). The solution is to take advantage of the extra function indexes. So if you want to have, let's say, 1024 possible float constants for a given symbolic regression application, you simply reserve 4 function indexes (4*256 = 1024) which point you to 4 different (1 line) float handling routines. For example, the third such routine might look like:

```
float Float3() {
   return FloatConstantTable[2][current_node->tableIndex] ;
};
```

13.6.3 The Jump Table Mechanism

Our jump table mechanism is simply the array of **`token`** objects. The primary benefit of such a jump table is that now we can select which function to execute with an array dereference as opposed to a **`case`** statement. Although compilers will often compile a **`case`** statement into such a jump table, it will still do bounds checking on the index, and so will be slower than the hand coded jump table. If **`current_node`** points to a **`Node`** object, and **`Tokens`** points to the array of **`token`** objects, then **`EvalNextArg()`** can be implemented with the statement:

```
return (Tokens[(current_node++)->functIndex].Funct)() ;
```

In other words, this highly efficient code fragment amounts to nothing more than incrementing and de-referencing a pointer, referencing an array and making a function call.

13.7 The Prefix, Jump-Table (PJT) Approach

We think the best overall implementation of the genome interpreter uses these 4 key concepts: a prefix ordering scheme, general data support, a 2-byte node representation, and a jump-table mechanism. It is the cleanest and most modular approach, since it avoids the need for **`case`** statements to determine the type of a node. It is the most flexible since it naturally handles all basic flow control constructs. Finally, in addition to the previous attributes, it is the most efficient approach in terms of its node space/time overhead. As any designer can testify, we were quite lucky to have such a clear winner.

13.8 Results

One must consider the relative importance of memory vs. speed. Koza [1992] provides empirical findings which demonstrate that by *tripling* the population size and keeping the case count constant, the overall efficiency of the simulation is usually *doubled* (the number of individuals needed to be processed is reduced by 1/2). Based on this empirical relationship, we can derive an approximation relating the efficiency (e) of one interpreter with respect to another in terms of their node memory size (m) and evaluation time (t):

$$\frac{e_2}{e_1} = \left(\frac{t_1}{t_2}\right)\left(\frac{m_1}{m_2}\right)^{\log_3 2} \approx \left(\frac{t_1}{t_2}\right)\left(\frac{m_1}{m_2}\right)^{0.631} \quad , \tag{13.3}$$

where efficiency is the simulation time required to have a 99% probability of convergence. Therefore, memory overhead is not as devastating as time overhead in the node implementation. An interpreter which evaluates nodes 4 times slower than an alternative interpreter but uses a node representation 1/9 the size, will be equally as efficient. We use Equation (13.3) to provide a general approximation of performance between the 5 different interpreter variants discussed above.

We tried our linear implementations out on a number of simple problems suggested by Koza [1992]. The results confirmed that we were able to converge as expected and essentially match the results of the established tree-based representation. This follows logically since we have a functionally equivalent system. Table 13.1 shows the performance results of 5 approaches presented in this chapter and also summarizes the advantages and disadvantages of each. We ran each test with 5 different compilers on 4 different platforms. The data shown below represents an average over the multiple runs.

13.9 Advanced Topics (Looking for Roadblocks)

Before we can claim that the PJT interpreter represents a generally usable implementation, we should first overview as many programming constructs which might be used in a GP setting, and see if such constructs pose any significant problems to the approach. Note that our focus in this discussion is on the mechanics of the implementation as opposed to the actual usage of such constructs in real simulations.

13.9.1 Beyond Closure: Handling Multiple Data Types

It is certainly possible that a given application has function nodes which work with a variety of different data types simultaneously. Such applications need an interpreter which allows all of these functions to work together in a transparent and flexible manner. C++ allows objects to overload their associated casting operators and constructors allowing automatic conversion between source and destination data objects. So, we can select a generic data-type which all of the other object types will convert to and from (using the automatic send/receive methods above) in a transparent manner.

This approach, which is more extensive than "closure" as defined by Koza [1992], works well provided that all of the different data types are convertible to/from the generic type. Montana [1993] discusses applications where the given data types include user defined structures like vectors and matrices. For such applications, the simple automatic-closure mechanism might not suffice since converting a **`vector`** to an **`int`** for example, might not make sense for certain handlers. Montana gets around the problem by providing special genetic operators which only allow certain connections to be made in the node structure. Thus a handler which expects an input of data type **`T`** will only be connected to handlers which output type **`T`**.

An alternative, is to have the genetic operators make connections based on a connection-strength table. For example, if a crossover method was to perform a subtree exchange resulting in a **`string`** to **`float`** connection, it could reference this table to see how viable that connection is. If the strength is less than some arbitrary value, then the operator simply looks again for a subtree which is more appropriate.

Table 13.1

Results and comparison of the different approaches discussed in this chapter. t_2/t_1 and e_2/e_1 are the run-time and effeicency, respectively, relative to the PJT approach, and m is the number of bytes each node occupies. "Virtual Function Tree" refers to coding the **`Eval()`** function as a vertual function, "If Statement Tree" refers to having a **`Type`** field parsed by a **`case`** statement, and "Function Pointer Tree" refers to storing a pointer to the evaluation function at each node. We assume **`MAX_ARGS`** is 2, and we disregard the memory for the **`ArgValues`** field.

Representation	t_2/t_1	m	e_2/e_1	Advantages	Disadvantages
Prefix/Jump Table	1.00	1	1.00	Most Efficient and Flexible Approach.	Can't do Parallel Evaluation.
Postfix/Mixfix	1.10	1	0.91	Memory Efficient, Good for Parallel Evaluation, Unconstrained Syntax.	Flow Control is Awkward.
Virtual Function Tree	0.92	4	0.45	Effective, Re-usable Code.	Size Overhead, Memory Manager Needed.
If Statement Tree	1.47	7	0.20	Conceptually Simple	Size Overhead, Slow, Memory Manager Needed.
Function Pointer Tree	0.81	6	0.40	Overall Fastest Approach, Conceptually Simple.	Size Overhead, Memory Manager Needed.

For certain applications, it also might be useful to have the interpreter pass parameters by reference as opposed to their value. The PJT scheme makes this easy since a node function can return a 2-byte address defined as {**`table:entry`**} which points to its return argument. The advantages of reference passing are threefold. First, functions can now return multiple arguments via structure references. Secondly, we now avoid the problem of passing the actual data of large data-structures (like a matrix) through the system stack thus avoiding unnecessary overhead. Finally, the address as defined above, allows the function-token to easily determine the data-type of the return argument (at run time) since the table value corresponds to the type. This run-time type information can then be used by the handler to convert arguments in a specific manner or to provide various services based on the receive types (allows the handler to overload itself).

Handling a set of mixed data-types does not seem to provide any roadblocks to any given interpreter implementation specifically. The PJT design does seem to provide a natural way to incorporate reference passing due to its table-driven emphasis. Overall, we feel that a more unified approach is needed and can be established as more difficult problems are attempted.

13.9.2 Module Implementation

This section discusses two aspects of modularization with respect to the linear PJT implementation. First we discuss a basic encapsulation technique and then we look into how modules are actually executed. One possible implementation of the module structure is:

```
class Module {
   int ArityMinus1;
   char ReturnType;
   int Size;
   Node *Genome;
} ;
```

Note that a **`Module`** has characteristics in common to both a **`Program`** (**`Size`**, **`Genome`**) and **`Token`**s (**`ArityMinus1`**, **`ReturnType`**). It is very important in a C implementation that all 3 of these structures be set up so that the module parameters correctly *overlay* the parameters it has in common with the other 2 structures. This overlaying allows us to treat a **`Module`** as a **`Program`** (during evaluation) or a **`Token`** (for referencing token description information) without any conditional coding needed. In C++, however, the *shadowing* mechanism used with multiple inheritance provides a superior approach to overlaying structures:

```
class Module : public Program, Token { /*...*/ };
```

Now we can have a polymorphic program pointer such that references like **`P->Genome`** are allowable where **`P`** can be a pointer to a **`Program`** or a **`Module`**. Furthermore, we can say **`T->ArityMinus1`** where **`T`** allows us to access the arity information in a **`Module`** or **`Token`** without having to know if **`T`** actually points to a **`Module`** or a **`Token`** object.

13.9.2.1 Encapsulation

There are various approaches for defining or encapsulating modules. Koza [1990] uses an approach called Automatically Defined Functions (ADFs). His approach is LISP based and involves every program being syntactically constrained. Angeline and Pollack [1993] suggest a free form approach where you simply encapsulate a subtree or a segment of a subtree which has already evolved in your current population.

Our technique follows from Angeline and Pollack. We first randomly select a subtree which is of a *minimum* size and inspect the terminals of that sub-tree. The number of different variable-terminals determines the arity of the encapsulated subroutine. Finally, we re-define each such variable-terminal in the subtree so that it references a general **`ArgStack`** instead of a global **`VariableTable`**. This then makes all of the parameters for that subtree “local”.

For an example, consider encapsulating the following expression:

Prefix: **`* + + X a X - Y b`** **Infix:** **`(X+a+X)*(Y-b)`**

If we were to compress the segment {+ + X a X}, we would have the following module defined after compression:

```
Module (M):
 arity = 1;
 size = 4;
 genome = {add, add, arg:0, const:a, arg:0}
```

The original program would also be compressed or re-defined such that the new second and third nodes now correspond to a “module-call”:

Prefix: **`* M X - Y b`** **Infix:** **`M(X)`*(Y-b)**

So now when the host program is executed, it will call the module-handler routine which will in turn find module **`M`** in the module table, and execute it’s internal genome sequence. When this happens, the variable **`X`** will now be processed as an argument. That is, the module handler will **`PUSH`** the value of **`X`** on the **`ArgStack`** before actually executing the module’s genome. Since, all of the variables in the module reference the **`ArgStack`**, the phenotype of the host program remains constant even though the genotype of the program has indeed been altered. Another point is that although the subtree contains 3 terminals, one of the terminals is a constant and the other 2 terminals are the same. Thus the arity of the newly defined module is only 1.

The encapsulation routine can be implemented with the following steps:

1. get a free module entry from the module table. If a free entry is not available, then abort.

2. find a random subtree of a minimum size from the host's genome array. Here we can use the same routine used to obtain subtrees in the crossover implementation.

3. inspect all of the terminals in the subtree and determine which ones are constants and which ones represent unique variables. The unique variables are re-defined to reference the **`ArgStack`** and **`ArgHandler`** as opposed to being variables.

4. define the module by copying over the subtree to the internal genome array of the module. Continue defining the module by setting the module's size to be equal to the subtree size and set the module's arity parameter to be the number of unique variables found in the subtree.

5. re-define the host program to reflect the compression/mutation operation. The subtree segment is replace by the call segment and the host-program's size parameter is decreased by the difference between the subtree size and the number of nodes in the call (this equals 1 + module's arity).

13.9.2.2 Module Execution

Module implementation with respect to the actual execution of modules seems like it might be easier to accomplish in a postfix implementation as opposed to prefix. This is due to the fact that subroutines naturally fit into a stack scheme. For postfix, this means that when a subroutine is to be executed, its parameters will automatically be waiting on the stack. For prefix, this is not the case. Each function or module is responsible for obtaining its own arguments. Since these arguments are outside the module, the module handler needs to load the **`ArgStack`** with the needed parameters on behalf of the given module before that module is actually invoked. Since such a handler can obtain the needed arguments by simply doing an **`EvalNextArg()`**, it turns out that our prefix scheme allows for a straightforward implementation and does not require any additional overhead compared to postfix. The code is shown in Figure 13.10.

13.9.3 Handling Recursion

Recursion can be promoted at two levels in a GP application. The first approach is to allow the main program itself to be called recursively by an internal handler. By adding a

```
inline unsigned char CurrentIndex() { return current_node->t ; }

void module_handler ()
{
   Program *PrgSave; Node *NodeSave; int i;

   // initialize the local arg stack
   for (i=0; i<ModuleTable[INDEX].Arity; i++)
   ArgStack.Push = EvalNextArg();

   // now save the global prg pointer and token index
   PrgSave = Prg; NodeSave = current_node;

   // now make our gloable refs point to the subroutine at hand
   Prg = ModuleTable[INDEX]; NodePtr = Prg->Genome;

   // now execute the module at hand
   EvalNextArg() ;

   // restore the global references
   Prg = PrgSave; NodePtr = NodeSave;
}
```

Figure 13.10
The module handler.This is called when `Eval`ing a module node.

`Recurse-Handler` to the function set, the program itself can essentially be "called". This `Recurse-Handler` would invoke the program in the same way that the platform invokes the program except that the program would be passed an argument from an internal node as opposed to a "case" value. For example, consider a simple symbolic regression problem of 2 independent variables. For such an application, each program is essentially a subroutine of arity 2. A `Recurse-Handler` could be implemented as a function-token, with an arity equal to the program's arity, allowing it to obtain the proper number of arguments before calling the host program. In the example code shown in Figure 13.11 we use the argument stack. Each time the host program is recursively called, its variable handlers will access values further down this stack. We also use a special recursion stack so we can return to this point when the program completes.

Now instead of invoking programs with the usual `EvalNextArg()` macro, we need to keep restarting the program as long as the recursion stack is non-empty at the popped `current_node` location.

The second recursion scheme involves the use of modules. The idea is to allow a program to contain recursive subroutines. This is accomplished by mutating an existing module `M` by finding an internal function node within `M` which has the same arity as `M` itself. We then replace this internal function node with a Module-Recurse-Handler node which will

```
RETURN_TYPE RecurseHandler()
{
   //first load up our argument stack with the calling parameters
   for (i=0; i<PROGRAM_ARITY; i++)
   ArgStack.Push (EvalNextArg()); //PUSH automatically

   //save our current location so that the host eventually
   //returns here
   RecurseStack.Push (current_node);

   //now restart the host program and "call" it
   current_node = Genome - 1; EvalNextArg();
}

EvalRecursiveProgram ()
{
   current_node = Genome -1; EvalNextArg(); //normal invocation
        //assist recursion:
   while (current_node = RecurseStack.Pop) EvalNextArg();
}
```

Figure 13.11
A handler for nodes which mean “call this program recursively”, and a wrapper to make the first (non-recursive) call.

“call” `M` in a similar manner that the handler outlined above called the main program. To maintain syntax, this Module-Recurse-Handler would have to able to assume an arity which is equal to the arity of `M`. For either recursive approach, we feel that it would be advantageous to bind the recurse-handler to an if-handler in order to increase the chances that the recursion process completes.

13.9.4 Simulated Multi-Tasking

Many applications require multiple trees to be evaluated in parallel. That is, one often wants to optimize something which consists of many parts in which each sub-part interacts with the other sub-parts in a parallel manner (i.e. the emergent behavior of ant colonies). Fortunately, it is not impossible to achieve multi-tasking with prefix. The basic idea is to somewhat combine prefix with a stack implementation. In other words, the genome ordering is still a prefix ordering, but the parameters are passed by an explicit stack and the genome is no longer executed by recursion but by the use of an execution stack which allows us to mimic what the system stack did. See Figure 13.12

13.9.5 Using Tables to Evaluate Diversity

The Jump Table mechanism results in a fair number of tables (arrays) being used for constants, variable, modules etc. An added bonus of a table-driven approach is that such

```
class Task { PROGRAM &prg; int Index, Status; };

main ()
{
   Task Tasks[2] = {{Prg[0], 0, RUNNING, {Prg[1], 0, RUNNING}};
   MultiTask (Tasks, 2);
}

multi_task (TASK &Tasks, int TaskCount)
{
   int t=0; //the task index

   for (;;) { // Forever
      switch (Functions[Tasks[t].Index].Type) {
      case FUNCTION:
         ExeStack.push (Tasks[t].Index); //save index for later exec
         Tasks[t].Index++; //goto the next node for this task
         break;
      case TERMINAL:
        //execute the terminal node which will push a value on stack
        EVAL (Tasks[t].Index++);

 //check if stack count is equal to the arity of the last func
 //on the ExeStack which is still waiting to execute. If it is,
 //then go ahead and evaluate this funct we saved earlier
         if (ArgStack[t].Count == ArityOfLastFunction)
            EVAL (ExeStack[t].pop);

 //now do a task switch
         if (++t == TaskCount) t = 0;
         break;
      case END:
         tasks[t].status = COMPLETED;
         if (all_tasks_completed ()) return; //****done*****
      }
   }
}
```

Figure 13.12
Simulated Multi-tasking using prefix.

tables can provide an easy means to dynamically evaluate diversity in the current population. This is accomplished by adding a "**`Used`**" bit to each entry in any of the tables just mentioned. To measure the ongoing diversity of the constants, we add a line to the constant handler:

```
void ConstHandler ()
{
 constTable[INDEX].Used = 1;
 return (constTable[INDEX]);
}
```

Now if we clear all of the "**`Used`**" bits to zero at the start of each generation, the bits which are not set when the generation has completed, correspond to constants which are no longer in use. We tried this out for our simple regression application and were amazed at how quickly the total number of unused constants increased.

13.10 Conclusion and Future Directions

The lower-level implementation issues surrounding what we call the Genome Interpreter have been presented. As opposed to just discussing issues along a single implementation direction, we have considered 5 different variations and evaluated each of these interpreters based on their efficiency, flexibility, and ease of coding. Our results clearly showed that a linear, prefix-ordered, jump-table approach (PJT) provides the best overall framework for the actual implementation.

One method of evaluation that could be even faster than a genome interpreter is a genome *compiler.* Although our interpreter has little time overhead, the overhead can still be significant if the functions and/or terminals take only a few machine instructions to implement. For example, when performing constant and variable lookup, and addition and multiplication, the overhead of the interpreter will still swamp the time needed to actually evaluate the node. However, the only way to do away with the function call overhead in an interpreter is to write many versions of each function, namely, one which does a function call when it's argument is a complicated subtree, and others to do in-line variable or constant lookup.

This could be solved using a genome compiler, which, given a tree, outputs machine language that doesn't perform any function calls. If, as is often the case, the same tree will be evaluated many times, the tree will only need to be compiled once instead of interpreted many times. Additionally, since crossover combines large chunks of different trees, we may be able to save compile time by not recompiling the chunks, but reusing the compiled code of the parents. One drawback, however, is that flow control may skip large parts of the tree, so that only a small part of the tree is ever interpreted whereas all of it must be compiled. This could make compiling slower than interpreting. And all sorts of variations are possible, such as compiling modules but interpreting the rest. Overall, attempting a genome compiler is a promising direction for speeding up evaluation even further.

Acknowledgments

Thanks to Greg Schmidt of Allen Bradley Controls for feedback on various issues presented in this chapter. Also, thanks to Graham Spencer of Stanford University for his early contributions with respect to testing. We also must recognize the importance of various discussions which took place on the GP internet mailing-list (organized by James Rice of Stanford University) related to implementation issues. Finally, thanks to those who reviewed this chapter for their many helpful comments.

Bibliography

Angeline, P.J. and J. B. Pollack (1993) Evolutionary Module Acquisition, *Proceedings of the Second Conference on Evolutionary Programming*.

Eckel, Bruce (1990) *Using C++*. Osborn McGraw-Hill.

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

Holland J. (1975) *Adaption in Natural and Artificial Systems*, Ann Arbor, MI: The University of Michigan Press.

Koza, J.R. (1992) *Genetic Programming*. The MIT Press.

Montana, D.J. (1993) Strongly Typed Genetic Programming, BBN Technical Report #7866, May 1993.

Tackett, W.A. (1993) Genetic Programming for Feature Discovery and Image Discrimination, *Proceedings of the Fifth International Conference on Genetic Algorithms*. Morgan Kaufmann..

Winfield, A. (1983) *The Complete Forth*, Wiley Press