GENETIC PROGRAMMING IN C++

(A manual in progress for gpc++, a public domain genetic programming system)

BY

ADAM P. FRASER

UNIVERSITY OF SALFORD,
CYBERNETICS RESEARCH INSTITUTE,
TECHNICAL REPORT 040

TABLE OF CONTENTS

GENETIC PROGRAMMING IN C++

(Gpc++ Version 0.40)

Adam P. Fraser

Department of Electronic & Electrical Engineering, University Of Salford, Salford, M5 4WT, UK
Phone: 061 7455000 x3633
Email: *a.fraser@eee.salford.ac.uk*

> *'Evolution is all about assembling the improbable by tiny steps;*
> *and not until the unlikely has been reached do we notice just what it can do.'*
> Steve Jones - 'The Language Of The Genes'

INTRODUCTION

This is a short report documenting the manual evolution of a computer program which evolves computer programs. Gpc++ from humble beginnings has evolved into a multiple purpose genetic programming kernel on which to evolve modules of program code.

This documents outlines the genetic programming paradigm and its extension, automatically defined functions, and their implementation within a steady state architecture. A number of selection, recombination and mutation operators have been produced and these are explained. The C++ class definitions are discussed and examples of evolving code are worked through to show the user how to write the code for themselves.

For a more thorough examination into genetic programming John Koza's books and Andrew Singleton's magazine article are an excellent beginning. For eloquent explanations on steady state genetic programming the reader may wish to read papers by Craig Reynolds. These references are outlined in more detail in the reference section at the end of this report.

GENETIC PROGRAMMING: AN OVERVIEW

Genetic programming is a further extension to the complexity of evolving structures. Within the genetic programming system the structures undergoing adaptation are hierarchical computer programs based on LISP-like symbolic expressions. The size, shape and structure of the solution as a genetic program is left unspecified and is found by using the genetic programming operators. Solving a problem therefore becomes a search through all the possible combinations of symbolic expressions defined by the programmer.

The processes which make up a complete run of genetic programming can be divided into a number of sequential steps;

1... Create a random population of programs using the symbolic expressions provided.
2... Evaluate each program assigning a fitness value according to a pre-specified fitness function which measures the ability of the program to solve the problem.
3... Using some predefined reproduction technique copy existing programs into the new generation.
4... Genetically recombine the new population with the crossover function from a randomly chosen set of parents.
5... Repeat steps 2 onwards for the new population until a prespecified termination criterion has been satisfied or a fixed number of generations has been completed.
6... The solution to the problem is the genetic program with the best fitness within all the generations.

The creation of a genetic program is the combination of the domain dependent symbolic expressions predefined by the designer. These symbolic expression are divided into two sets, if the expression requires arguments it is placed in a function set otherwise it is placed within the terminal set. Each individual expression must have the property of closure and some combination must be sufficient to solve the problem.

The LISP symbolic expressions that are the basis of the genetic programming system are best shown graphically with a parse tree. The symbolic expression (+ (* 3 (- 4 2)) (+ 2 2)) becomes the parse tree in Figure 1. The functional alphabet of the genetic program is { +, *, - } and the terminal alphabet is { 2, 3, 4 }. Though this alphabet is correct for the genetic program this is probably only a small subset of the total available to the genetic programming system. Genetic programming also attempts to measure the complexity of the solution by giving each genetic program a structural complexity. In the case of the example shown the structural complexity is 9 and is simply a measure of the nodes within the parse tree.

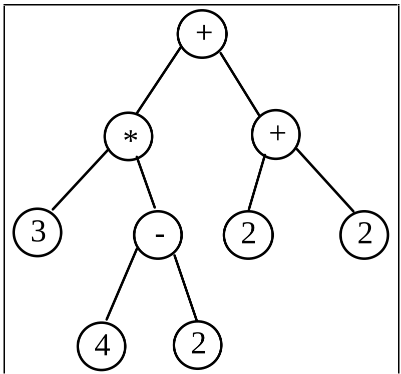

Figure 1: A Genetic Program

Closure

The symbolic expressions will be operated on by the genetic programming mechanism it is therefore necessary that all possible arrangements of the expressions will lead to a program which can be evaluated without error. This property of the expressions is termed the closure property.

The standard division operator (/) is an example of a possible infringement of the closure property. If within the evaluation process there is a division by zero the evaluating program itself will cause an error which will halt the genetic programming system. It is therefore necessary to rewrite the division operator in a way that could not cause a halt in the evaluation process.

```
int divide( int x, int y ) {   if  ( y ==  0 )  return 0;      else  return ( x / y );  }
```

Most genetic programmers who use the divide operator write this as %.

Sufficiency

When choosing the symbolic expressions it is important that they can express in some combination a solution to the problem. Genetic programming makes obvious the a priori knowledge being placed within the evolutionary mechanism by making it part of the design process.

This explicit knowledge by the designer is a necessary prerequisite of any evolutionary design but does not limit the use of the system. It is only at the point where detailed knowledge of the problem and how to solve it are placed within the evolutionary mechanism that limitations can occur if novel solutions are required.

For example in a problem of the solitary trail following ant, in which an artificial agent evolves the ability to follow a trail which increases in complexity the further the agent travels, the symbolic expressions contain IfPheremoneAhead, MoveForward, TurnRight and TurnLeft. If the MoveForward expression was excluded from the evolutionary mechanism then the solution could not be found as it is obvious that no other operator moves the agent in any direction.

AUTOMATICALLY DEFINED FUNCTIONS

Koza spends a whole book explaining and analysing automatically defined functions (Genetic Programming II or Jaws II: The ADF Strikes Back) so it is unlikely that this short introduction will do more than wet the appetite of the reader.

Genetic programming can be considered as attempting, through selectionist techniques, to produce computer programs which map the environment (which also defines the problem). If we consider a chess board and a pawn which can move one space in any direction at any time then a genetic program, whose fitness is proportional to the amount of new squares moved to, would have to make at least 64 moves to gain the maximum fitness. Within such a structured environment it is obvious that there are certain repeatable blocks of moves which would have the same effect. For example if the pawn was started from the bottom left hand corner of the board it could move north seven times (as we start on a block), east once, south seven times and east once again. Do this four times and the pawn has completed its task. If the genetic program could reproduce this it could go from a length (or structural complexity) of 64 to (7+1+7+1) +4 = 20. This is the promise of automatically defined functions.

Rather than using a single root branch of the genetic program the system allows further branches to be added. Each branch has a separate function and terminal set with the root branch commonly being used as a simple function call for the other branches (for further explanation see the Lawnmower problem section). Recombination is constrained to act within particular branches only as the differing function and terminal sets could cause conflict of the closure property. The genetic programs can be considered as having co-evolving branches.

STEADY STATE GENETIC PROGRAMMING

The genetic programming paradigm as detailed by Koza uses two populations for the current and next generation. Reproduction copies members of the current population into the next generation by a selective criteria. Members within this new generation are then recombined to produce new genetic programs. This next generation becomes the current and the process continues until the system finishes.

The initial operating system for gpc++ was DOS which has a very limited amount of memory available to the programmer without resorting to confusing (and non-portable) methods. The code for gpc++ therefore had to limit the size of the memory allocation used by the system. The simplest method was to dispense with the two populations and use a slightly different method for reproduction. This method has since been given a name, steady state genetic programming.

In steady state genetic programming the parents to be recombined are selected from the population again using some criteria but a child is also chosen from the same population. The recombined genetic program is evaluated and then takes the place of the child already selected. A generation within such a system is considered as completed once the number of children created is equal to the size of the population.

PARSE TREES

The basic premise for the implementation of gpc++ is that the important detail of GP is in the evaluation. The creation, selection and genetic operators are assumed to take a negligible amount of time so instead all optimisation should be focused upon limiting the processing overhead of evaluating a genetic program. This assumption has been borne out in the experiments performed in gpc++ where the evaluation time has steadily increased with the population staying approximately constant.

The simplest and fastest method of moving from one part of the genetic program to another would be to have each memory location laid out concurrently. Unfortunately the hierarchical nature of genetic programming makes this process impossible. The second fastest method is to keep pointers to the next block of code which you wish to travel to and jump from memory location to memory location at will. This is the implementation used within gpc++ which is based on the basic structure of a parse tree.

CREATION

The type of creation performed can be one of five types which are defined in the code as numerical values;

> Variable: Where the genetic program can be of a size or structure up to the maximum depth specified for creation (designated as 0).
>
> Grow: Where the creation mechanism can only choose functions until the maximum depth is reached when a terminal must be chose. This causes the size and structure of the genetic program to be the same in all random creations (designated as 1).
>
> Ramped: Either variable or grow. This changes the standard methods so the population is broken into smaller blocks and the creation mechanism attempts to produce genetic programs with increasing possible depths up to the maximum depth for creation. For example, a population of ten with a maximum depth of 6 using the ramped grow method would have 2 members of the population with depths of 2, 2 members of the population with depths of 3 and so on up to a depth of 6 (designated as 3 for ramped variable and 4 for ramped grow).
>
> Ramped Half and Half: This is the creation mechanism used in the majority of the genetic programs developed in Koza's Genetic Programming. The algorithm permits half the population to be created with ramped variable and the other half to use ramped grow (designated as 2).

The type of creation mechanism can be chosen using the CreationType contained in the GPVariable structure. The definitions of these creations types can be found in the file *gp.hpp* and are accessed by the user through the *gp.ini*. The code for the creation of a genetic program can be found in *create.cc* and for the population in *pop.cc*. All decisions about the type of creation is made at the population level.

SELECTION OF GENETIC PROGRAMS

Within the genetic programming system the time comes when from the population some genetic programs must be selected. This is produced through *SelectParentsAndChild()* which is called within *generate.cc*. This function as its name suggests selects the parents for crossover and the child to place the result into. It is designed such that the user can make a decision between which type of selection method to be used. The standard is a fitness probabilistic method (the roulette wheel) which is included for historical reasons. The favoured method of most genetic programmers as gauged from discussions on the genetic programming mailing list is tournament selection. Demetic selection allows the genetic programmer to alter the population dynamics of an evolving system which in certain cases is a useful tool.

Fitness Proportionate

Fitness proportionate selects the best genetic program using a probabilistic method based mathematically on the roulette wheel of a standard genetic algorithm. For a genetic program with 20% of the total fitness, if 100 were chosen there would be an expected 20 of that genetic program. The nature of this selection method means that for a small number this will not be the case and that some genetic programs with a relatively small fitness will have a minimal (but still significant) chance of being chosen. Once the probable best genetic program has been found this replaces the worst which is selected with an 'inverse' fitness proportionate method. The code for fitness proportionate selection can be found in *probable.cc*.

Tournament Selection

Tournament selection randomly selects a number of genetic programs from the population. The fitness of each member of this group are compared and the actual best replaces the worst. This method is one which can be easily implemented using a parallel methodology. The number to randomly select for each group is currently set at 5 and can be easily changed. The code for this selection method can be found in *tourn.cc* as is the definition of the tournament size.

Demetic Grouping

Demetic grouping has been predominately used as a method for halting premature convergence of a particular artificial evolutionary technique without knowledge of the problem domain.

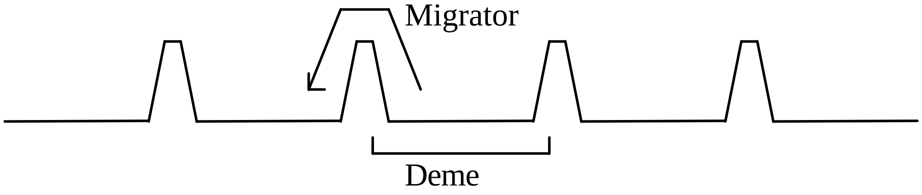

Figure 2: A One Dimensional View of Demetic Grouping

The total population undergoing the evolutionary process is subdivided into a number of groups (demes) which are unable to interact with other groups except through the use of migratory agents which are selected using a probabilistic measure (Figure 2). This subdivision of the population allows the demes to evolve along separate paths without this path becoming tightly focused upon any particular area within the global search space. This code can be found within the *deme.cc* file and there are a number of definitions notably those of probabilistic wanderer selection and size of the demetic group.

CROSSOVER

Subtree Crossover

Crossover selects two genetic programs from the population and selects one point on each. Each sub-tree from this point is swapped from the other. The closure property of the genetic program ensures that the these new genetic programs are still 'legal' possibilities within the domain. The code for crossover can be found in *generate.cc* which acts at the population level with the code within *cross.cc* acting upon individual genetic programs which unfortunately still requires knowledge of the genetic program structure.

Ad Addendum

Recently new methods of crossover have been discussed which take two or more points on the genetic program and cross between these points. This is relatively difficult to code as selection of a cut point within the same sub-branch of the genetic program is not a simplistic process. Also the improvement such a technique can have to the genetic programming paradigm is not immediately apparent. If anyone wishes to add this or other types of crossover the file *cross.cc* should be altered.

MUTATION

Allele Mutation

This comprises of genes within the genetic program being swapped with other genes with certain constraints. Any terminal can be swapped with any other terminal but functions can only be swapped with other functions with the same number of arguments. This means that the mutation does not have to create new branches when different function types are swapped which would probably slow any form of convergence if the mutation rate has been set relatively high. This can be found in *allelem.cc*

Shrink Mutation (Alpha Version - See Note)

Shrink mutation (which, I think, was an idea by Andrew Singleton and whose article in BYTE magazine Feb. 1994 gives further explanation) takes the child of a particular gene and moves that child into the position of the parent. This means that genetic programs will 'shrink'. This is a particularly useful property when considering how long some genetic programs get as the evolutionary process continues.

NB: The shrink mutation code at the moment seems to only work on particular compilers and with particular problems (most ADF problems work, others seem intermittent). For this reason *shrinkm.cc* has not included in the gpc++ package. If you are a C++ hacker or think you desperately need *shrinkm.cc* please contact the author or wait for the next version.

GENETIC PROGRAMMING INITIALISATION

Once the genetic programming run has begun the system attempts to read in a file called *gp.ini*. If this file does not exist the system creates a default and then exists. The *gp.ini* should always be checked before running the code as incorrect parameters cannot be checked within the code and may cause the program to crash (the ADF parameter is particular susceptible to being forgotten about as I know from bitter experience).

As already shown a default *gp.ini* will look something like the file shown below. The variable names are self explanatory the numerical values for the creation type can be found by looking at the Creation section of this document and also in *gp.hpp*. If you wished to use Koza's standard of ramped half and half the setting would be 2. The MaxFitness is the maximum fitness of the genetic program and is used within the probabilistic fitness measure to calculate the worst in the population. If the maximum fitness is unknown use a different type of selection method. The MaxCreation and MaxCrossover variables are the maximum depth at creation and crossover respectively the defaults are those used by Koza.

The ADF variable is very important if you wish to have only a single branch within the genetic program this is designated as the root branch and the variable should be set to 0. Every further branch is designated as an ADF and must be place in a variable hence in the lawn mower problem to be discussed later in this report there is a root and 2 automatically defined functions the ADFs variable must be set to 2.

The mutation rate is by default set to 0 giving no mutation at all. If this value was set to a 1000 it would designate that in all probability one genetic program in a 1000 would be mutated. A value of 1 would require every genetic program in the population to be mutated.

```
// default gp.ini <--- this line is not important
            CreationType   : 0
            Evaluations    : 100
            MaxFitness     : 100
            MaxCreation    : 6
            MaxCrossover   : 17
            ADFs           : 0
            Mutation       : 0
```

CLASS DEFINITIONS FOR GENETIC PROGRAMMING

In attempting to implement the genetic programming system a class hierarchy similar to that shown in Figure 3. In such a hierarchy all genetic operation available to the user act upon the population and all underlying definitions and operations should be invisible. The need to produce evolving code and other operators means that a limited exploration of the underlying class definition are necessary.

Population

The population is primarily made up of two components an inherited class structure of GPVariables and a pointer to the first member of a list of genetic programs which occupy a consecutive group of memory locations. Three unsigned long variables contain the information of the length, fitness and depth of the complete population (the inclusion of a total depth parameter is made as further analysis of GP may lead to this becoming an important parameter of the complexity of a particular population).

```
class Population : public GPVariables
{
       GP *pgpHeader;                          // pointer to the first member in the gp list
       unsigned long  uliFitness,            // fitness of total pop
                      uliLength,             // total length of pop
                      uliDepth;              // total depth of pop not currently used
};
```

GPVariables

The GPVariables defines all the parameters of the genetic programming system and are given on the command or through the gp.ini file. The variable names are hopefully self explanatory.

```
struct GPVariables
{
  unsigned int        PopulationSize,
                      NumberOfGenerations,
                      NumberOfADFs,
                      CreationType,
                      NumberOfEvaluations,
                      MaximumFitness,
                      MaximumDepthForCreation,
                      MaximumDepthForCrossover,
                      NumberToMutate;
   unsigned long     MaximumSumFitness;
};
```

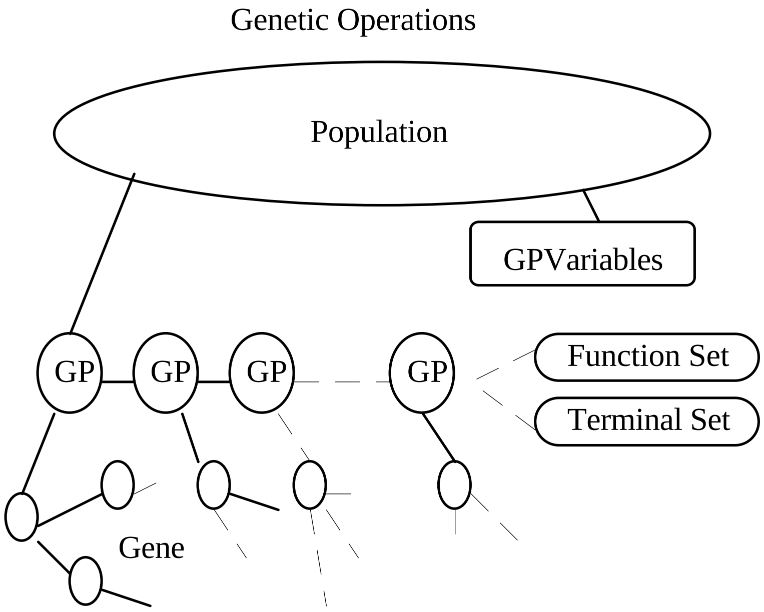

Figure 3: The Classes For Genetic Programming

GP

The Genetic program definition is of type struct as the members need to be publicly accessible to other functions. Two variables of unsigned int define the length and fitness of a genetic program. The structure also has a further member which points to first of a list of pointer to genes. Each gene is the start of a branch of the genetic program. In a program with only a root the *ppgHeader would point to a list of only a single gene. At creation the genetic program structure accesses the information contained in the FS (FunctionSet) and TS (Terminal Set) structures, this can be found in create.cc.

```
struct GP
{
//Pointer to a pointer of genes which are the headers for each tree of the adfs
       Gene **ppgHeader;
// the fitness and length of a particular genetic program
       unsigned int iFitness;              // if fitness gets over 65535 oh dear
       unsigned int iLength;                // If length gets over 65535 oh dear
};
```

Gene

The gene structure will be familiar to any programmer who has developed code using a tree structure. Each gene can have either a child or fellow neighbours which are defined as pointers to further genes. The unsigned int member iValue is the value of gene which is translated in the evolving code into the appropiate procedures.

```
struct Gene
{
// the value of gene, gpcpp deals only with numerical values
       unsigned int iValue;       // if more than 65535 program blocks, change!!!
// pointers to child gene (if a function) and next gene (if part of a function arguments)
       Gene    *pgChild,
               *pgNext;
};
```

PRODUCING EVOLVING CODE

In designing this architecture there has been attempt to allow the programmer to be able to use either C or C++ to write evolving code. This has been only partly successful as writing code still involves knowing some of the structure of the underlying architecture and hence C++. Hopefully with the examples given a C programmer can just use the code and techniques found there to write their own. If problems are found then please contact the author.

Five functions are crucial to the compilation and linking of the genetic programming system these are;

```
               ostream& (*TranslatePrint)( ostream& os,Gene *pg);
               unsigned int EvaluateFitness( GP *pgp, int Evaluations);
               void InitialiseGPS( void );
               void CleanUpGPS( void );
               ostream& operator <<(ostream& os, GP *pgp )
```

One further function which is used throughout the code is

```
               FITNESS (*Translate)( Gene*)
```

this is not called by gpc++ but by the functions above. The FITNESS is simply a definition to a particular type of data structure.

SYMBOLIC REGRESSION

The code shown below is a definition of these functions for the symbolic regression problem included with gpc++ (*symbreg.cc*). The symbolic regression problem attempts to find a function which fits a close as to that prespecified by the programmer. It does this by taking a number of points on the functions curve (in this case 10) and evaluating the genetic program with the value X set at that particular value. The accumulated difference between the two set of values should tend to zero as the genetic program gets closer to the function. In the example code this function is simply $x^4 + x^3 + x^2 + x$. The FITNESS is defined as a type float which is a 32-bit value in C/C++.

```
// In gpcpp all function and terminals are considered as numbers this tells the system
// what those number should be so they can be use in Translate...() functions
void InitialiseGPS()
{
// F(main) = { *,+,-,% }
// T(main) = { X }
       if (!(FunctionSets[0] = new FS( 4, 1,2,3,4, 2,2,2,2 )) ) ExitSystem( "Initialise");
       if (!(TerminalSets[0] = new TS( 1, 5 )) ) ExitSystem( "Initialise" );
// only need to be set up once so use global ....
       Translate = TranslateROOT;
       TranslatePrint = TranslatePrintROOT;
// Run through the function and values working out answers to function defined at
// the beginning of this file in range of 0 -> 10...............................
//  FUNCTION = X^4 + X^3 + X^2 + X..............................................
```

```
        for ( int i = 0; i < 10; i++ )
        {
                ques[i] = (float)i;
                answ[i] = FUNCTION( (float)i );
        }
}

//This is called right at the end of the GP system and can clear up all global variables
// created in InitialiseGPS() and anywhere else..........................................
void CleanUpGPS()
{
        delete FunctionSets[0];
        delete TerminalSets[0];
}
```

InitialiseGPS() is called at the start of the genetic programming run. Within InitialiseGPS() the user can define any global variables which are not going to be changed in the evaluation process. In the example the question and answers for the symbolic regression problem are calculated. The major use for InitialiseGPS() is to define the function and terminal sets. These must always be called FunctionSets[...] and TerminalSets[...] as this is the names searched for by create.cc in creating the genetic programs. The function and terminal sets are defined as a set of numbers. For terminals the initial value is the number of terminals and the next values are the terminals **unique** value. Functions use a similar system but include a further set of values for the number of arguments each function has to take. Therefore the function + (designated as the value 2) takes two arguments which could either be a further function or the terminal X (designated as the value 5).

CleanUpGPS() is the partner process of InitialiseGPS() and is called at the end of a particular run. This function deletes any globally allocated memory and the function and terminal sets.

```
// The translate function for the procedural calls from GP iValues
FITNESS TranslateROOT( Gene *pg )
{
        switch ( pg->iValue )
        {
// The multiplier values..........................................................
                case 1: return Translate( pg->pgChild ) * Translate( pg->pgChild->pgNext );
// the summation values...........................................................
                case 2: return Translate( pg->pgChild ) + Translate( pg->pgChild->pgNext );
// the subtraction values.........................................................
                case 3: return Translate( pg->pgChild ) - Translate( pg->pgChild->pgNext );
// divide is a special operator in GP (no closure property) so is somewhere else.
                case 4: return Divide( pg->pgChild );
// only one terminal the X variable which is set in evaluatefitness()
                case 5: return globalX;
// the default which will never be called as this code works......................
                default:return 0.0;
        }
}

// The translateprint function for the character strings from GP iValues
ostream& TranslatePrintROOT( ostream& os, Gene *pg )
{
        switch ( pg->iValue )
        {
// FUNCTIONS
// The multiplier values..........................................................
                case 1:         os << " ( *";  break;
// the summation values...........................................................
                case 2:         os << " ( +";  break;
// the subtraction values.........................................................
                case 3:         os << " ( -";  break;
// special form of divide which has closure traditonally shown as a %.............
                case 4:         os << " ( %";  break;
// TERMINALS
// only one the X variable which is set in evaluatefitness()
                case 5:         os << " X";    break;
// the default which return an error and will never be called......................
                default:        os << " Error";        break;
        }
        return os;
}
```

The implementation of gpc++ does not deal with the functions and terminals themselves but with integer values which are contained within the gene data structure as iValue hence the pg->iValue of the above code. Each value must be translated by the genetic program system. This had previously been developed through the use of Translate(Gene*) and TranslatePrint(ostream&, Gene*) modules. With the automatically defined function extension the translation mechanism is altered dependent on the branch of the genetic program the system is translating. Instead, therefore, of using the Translate...() functions pointers are used to these functions FITNESS (*Translate)(Gene*) and ostream& (*TranslatePrint)(ostream&, Gene*). As the symbolic regression does not actually make use of ADFs these pointers need only be specified once in the InitialiseGPS() system.

```
// The evaluate of function generally the most difficult to define for a problem
unsigned int EvaluateFitness( GP *pgp, int Evals )
{
        FITNESS rawfitness = 0, diff = 0;
// set up global genetic program variable to use ROOT macro defined above
// this is useful for ADFs but not particularly helpful here
        pgpGlobal = pgp;
// this next line is included just to stop any warnings in compilation
// it makes no difference to the code and can be deleted
        Evals--;
// the evaluation function checks with 10 values of the mathematical function
        for ( int i = 0; i < 10; i++ )
        {
// set up X variable for mathematical function.......................
                globalX = ques[i];
// calculate difference between the genetic program and the actual answer
                diff = fabs(answ[i] - Translate( ROOT ) );
// if this is really big don't make it to big...............................
                if ( diff > 100 ) diff = 100;
// add this difference to total rawfitness................................
                rawfitness += diff;
        }
// in this case the higher the rawfitness ( or accumulated differences ) the
// lower the fitness hence the next line.............................
        return (1000 - (unsigned int)rawfitness );
}
```

The EvaluateFitness() function is the problem the user wishes to solve. This problem is evaluated and a fitness allocated this is the return value which always has to be the type unsigned int though within the evolved code the fitness can be any type. In the example the pointer to a genetic program is set to a global variable genetic program (pgpGlobal) this is simply to allow the ROOT macros further on in the code. Automatically defined functions have led to the translation mechanism becoming less than elegant so macros are defined and used. The ROOT macro is equal to *(pgpGlobal->ppgHeader) which is in English the pointer to the first gene within the first branch of the genetic program pgpGlobal (see Macros section for further information).

The next section of important code is the problem. A range of values are considered from 0->9 in steps of 1. Each is posed as the question to the genetic program by setting the globalX to that value. This globalX is the only terminal in the genetic program. The answer from the genetic program is then compared to the actual answer and the difference between the two is the rawfitness.

At the final stage the rawfitness is the accumulated difference between the actual answers and genetic program's. The fitness we actually require is the similarity between the two answers not the difference and so the final return value is subtracted from the maximum possible value. Therefore a genetic program which returns identical answers to the function will have an accumulated rawfitness of zero but a returned fitness of a 1000 the maximum possible.

```
// Unfortunately because of ADFS and the need to alter the printing style of GPs
// the GP operator << needs to know reside here.................................
// This does not need to be understood just used by example if you wish..........
ostream& operator <<(ostream& os, GP *pgp )
{
        if ( pgp )
        {
// set up global genetic program variable to use ROOT macro defined above
// this is useful for ADFs but not particularly helpful here
                pgpGlobal = pgp;
// prints out a GP simple really isnt here. The initial bracket is a hack to get
// the total number of opening and closing number of brackets right............
                os << "(" << ROOT << endl;
        }
// must return this value even though it isnt really needed....................
        return os;
}
```

In printing the English version of a genetic program automatically defined functions have again complicated the code and the output stream operator needs to be contained within the evolving code. This operator takes a pointer to a genetic program as an input and also a reference to a stream. The genetic program is checked to see whether it exists and then the macros are used to print the desired branch(es). There is no real need to understand how this function works (though C++ streams are wonderful things) as there are enough examples to show how this code operates and to develop your own.

SANTA FE TRAIL / ARTIFICIAL ANT PROBLEM

The Santa Fe Trail is an artificial simulation of a trail that recreates some of the possible problems faced by a real ant attempting to follow the trail (Figure 4). The trail is placed onto a toriodal grid where an artificial agent walking off the right edge will walk back onto the left edge and similarly with the top and bottom edges. The files for this problem can be found in the ant directory if the compressed file was opened correctly if not the files *gpant.cc*, *ant.cc* and *trail.cc* must be included in the compilation. These files call two header files *ant.hpp* and *trail.hpp*, the file *ant.ini* can be renamed for the *gp.ini* of this problem.

The complexity of the trail increases depending on how far along the trail an agent has travelled. The first problem the agent faces is the ability to make a turn when no more pheromone can be detected in front of it. Once this simple problem is solved the agent must then evolve the relatively more complex problem of how to cross the gap within the trail. These problem become more difficult with the ant having to at one point evolve the ability to make a knights move, two steps forward and one step sidewards still without detecting a pheromone.

The complex problems for the ant are mirrored by the simplicity of the fitness function for the trail. The fitness of an agent is quite simply the number of new blocks with a pheromone scent stepped on by the agent. To facilitate this once an agent steps on a block with a pheromone this is dissipated, an ant should not get a high score for moving back and forth on the same piece of trail! The number of steps the agent can possibly take before it must finish was set to 600 though the total fitness score possible was only 91.

In summary, for an agent to solve the Santa Fe trail is must evolve the ability to move forward when it can detect the trail and to search for the trail when it cannot.

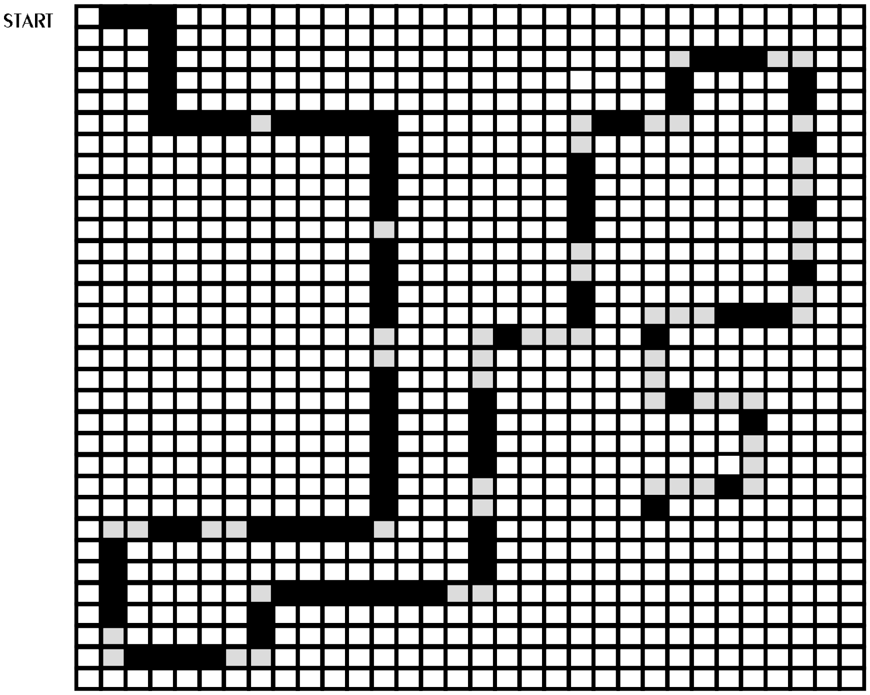

Figure 4 The Santa Fe Trail Grid

Representation Of An Artificial Ant

The artificial ant must be able to detect pheromone and to move in varying directions within the environment. The number of directions possible to the ant within the environment is limited to 4 so the behavioural functions of the system are simplified

The ant must, obviously, be able to move forward when on the trail and be able to turn towards the left or right the trail turns in wither of those directions. These expressions require no arguments and therefore the terminal set of the artificial ant is simply specified as;

```
               T( Trail Follower ) = { MoveForward, TurnRight, TurnLeft }
```

The function set is equally simple to specify as the only decision which the ant must make concerns the absence or not of a pheromone directly in front of the ant. The standard functions LISP function Prog2 and Prog3 are included to increase the sequential complexity available to the solutions evolved. The function set is specified as;

```
               F( Trail Follower ) = { IfPheremoneAhead, Prog2, Prog3 }
```

Therefore the InitialiseGPS() function is specified as below. While the world can be created globally the trail will be changed through each evaluation process and so is created in the EvaluateFitness() function. The FITNESS defined type in this problem is simply an int.

```
void InitialiseGPS()
{
// F( main ) = [ IfFoodAhead, Prog2, Prog3 }
// T( main ) = { TurnRight, TurnLeft, MoveForward ]
   if ( !( FunctionSets[0] = new FS( 3, 1,2,3, 2,2,3 ) ) ) ExitSystem( "InitialiseGPS()" );
   if ( !( TerminalSets[0] = new TS( 3, 4,5,6 ) ) )          ExitSystem( "InitialiseGPS()" );
// creates the world in which the ant resides. The trail is created seperately for each
evaluation process
   CreateWorld();
// note how as these stay constant they can be placed in the initialise block
   Translate = TranslateROOT;
   TranslatePrint = TranslatePrintROOT;
}

unsigned int EvaluateFitness( GP *pgp, int Evals )
{
// set the rawfitness to 0
   FITNESS rawfitness = 0;
// set up global variable for ROOT macro
   pgpGlobal = pgp;
// As ant 'eats' trail as it is evaluated we need to create a new one each time
   CreateTrail();
// Reset the position of the artificial ant with the energy which is equal to Evals
   ResetAnt( Evals );
// while the ant still has energy evaluate it accumulating the fitness
   while ( glbAnt.Energy > 0 )       rawfitness += Translate( ROOT );
// in this case the rawfitness is the fitness of the individual
   return (unsigned int)rawfitness;
}
```

Designing Functions

Functions implicitly take arguments and a knowledge of the architecture is needed. As the examples show below (from ant.cc) functions are sent a gene which is part of the genetic program. The gene sent has exactly the correct brethren for the function i.e. a gene which is sent to Prog3 will have a neighbour (pgNext) and it will also have a further neighbour (pgNext). For Prog3 each branch is evaluated and the fitness summated and returned the function IfFoodAhead() returns the value of a particular branch dependant on the return value of the boolean-like function IsFoodAhead().

```
// Prog3 is a C/C++ port of the LISP function which summates branches and returns
// the results. It is used to produce greater variety and ability in the genetic program
FITNESS Prog3( Gene *pg )
{
        FITNESS sum = 0;
        sum += Translate( pg );
        sum += Translate( pg->pgNext );
        sum += Translate( pg->pgNext->pgNext );
        return sum;
}
```

```
FITNESS IfFoodAhead( Gene *pg )
{
// if there is a food in the direction you are facing evaluate the first branch
       if ( IsFoodAhead() )   return Translate( pg );
// otherwise evolve the second branch.................
       else                   return Translate( pg->pgNext );
}
```

Designing Terminals

Terminals must return a fitness (though this may be 0) and cannot take parameters. In writing terminals the programmer can write code which produces side effects to the state of the environment or program. In the example shown below (again taken from ant.cc) *TurnLeft()* changes the direction of the ant (which is defined globally as glbAnt) as a side effect of being called, the return value in this case is 0.

```
// This is a function which operates only by side effects on the ant...............
int TurnLeft( void )
{
// check to make sure there is energy left.......................................
       if ( glbAnt.Energy == 0 ) return 0;
// turn left by adding 1 to moving and keeping within constraint of 4
       glbAnt.Moving = ( glbAnt.Moving + 1 ) % 4;
// decrease the energy of the ant...............................................
       glbAnt.Energy--;
// turning can not add to the fitness so return a 0.............................
       return 0;
}
```

LAWNMOWER PROBLEM

The paper which documents this problem is in 'Advances in Genetic Programming', Kim Kinnear (Ed) as a chapter called 'Scalable Learning in Genetic Programming using Automatic Function Definition' by John Koza. I would like to thank John Koza and James Rice for the copy of the paper from which I have developed the code (in one evening between 10pm and 1am !). The files for this problem can be found in the *lawn* directory if the compressed file was opened correctly if not the files *gplawn.cc*, *mower.cc* and *lawn.cc* must be included in the compilation. These files call two header files *mower.hpp* and *lawn.hpp*, the file *lawn.ini* can be renamed for the *gp.ini* of this problem.

Consider a lawn upon which we have placed an autonomous mower which must evolve the ability to mow the lawn. The lawn in question is a sixty four grid toriodal world. Using standard GP the robot would have to evolve sixty four different moves but by using automatically defined functions the repeatable structure of the environment can be used to produce shorter genetic programs. This type of problem is analogous to the chess problem with which automatically defined functions were explained but has the potential to do something useful.

The root branch of a genetic program can call its ADF0 branch as a terminal and its ADF1 branch as a function which takes a single argument. An example of a root branch is (ADF1 (ADF1 (ADF0))). The FITNESS definition for the functions and terminals of the system is a vector which can has a range of 0 to 7 i.e. [3,5]. The functions available to the branches are: V8A the vector addition of two arguments which returns a value which had the modulus by 8 operator applied to keep it within range, PROG2 the standard LISP operator and FROG which takes one argument and 'hops' to the vector given. The terminals are: LEFT turning the mower left, MOW which just mows forward one move and a set of all vectors possible within the ranges given. ADF1 also has a further possible terminal. the value returned by its argument which in this case will always be ARG0. FROG, MOW and LEFT all decrease the 'energy' of the mower and count as one step in the evaluation process.

The InitialiseGPS() and CleanUpGPS() functions of the system are simple to code once this information has been ascertained. Note that there are now three function and terminal sets for each branch of the genetic program and the unique values of the function and terminal only need to be unique within an individual branch. The production of a set of real numbers is produced using the RandomReal option in the terminal set creation the value after this is the range of real values the user requires (64). These values are stored in the values above 32768 to stop conflict with the terminal and function set values.

```
void InitialiseGPS()
{
// F(main) = { ADF1  , 1 argument }
// T(main) = { ADF0 }
        if (!(FunctionSets[0] = new FS( 1, 1, 1 )) ) ExitSystem( "Initialise");
        if (!(TerminalSets[0] = new TS( 1, 2 )) ) ExitSystem( "Initialise" );

// F(adf0) = { V8A, PROG2, 2 argument each }
// T(adf0) = { LEFT, MOW, Real }
        if (!(FunctionSets[1] = new FS( 2, 1,2, 2,2 )) ) ExitSystem( "Initialise");
        if (!(TerminalSets[1] = new TS( 3, 3,4,RandomReal, 64 )) ) ExitSystem( "Initialise" );

// F(adf1) = { V8A, FROG, PROG2 , 2,1,2 arguments respectively }
// T(adf1) = { ARG0, LEFT, MOW, Real }
        if (!(FunctionSets[2] = new FS( 3, 1,2,3, 2,1,2 )) ) ExitSystem( "Initialise");
        if (!(TerminalSets[2] = new TS( 4, 4,5,6,RandomReal, 64 )) ) ExitSystem( "Initialise"
);
}

// Now have to clean up 3 Function and Terminal Sets....
void CleanUpGPS()
{
  for ( int i= 0; i < 3; i++ )
        {
                delete FunctionSets[i];
                delete TerminalSets[i];
        }
}
```

The EvaluateFitness() is unusual in this problem as the evaluation of the genetic program does not return a value which can be used as a fitness. The fitness is instead calculated in the last line by running a piece of code which counts the number of grids with grass still in this value is subtracted from the maximum blocks of grass (64). The Mower in the code is a global variable which is acted on by the operators of the system.

A further important concept in designing evolving code is the consideration of what genetic programs can be produced with the function and terminal sets provided. In this case if the genetic program is made up entirely of V8A functions the program will continue evaluating in an infinite loop as the mowers 'energy' is only decreased by actually moving. Hence the second line within the evaluation loop which guards against this eventuality.

```
unsigned int EvaluateFitness( GP *pgp, int Evals )
{
// set up global variable necessary to get at ADF
        pgpGlobal = pgp;
// Reset the Mower........
        Mower.x = 4;
        Mower.y = 4;
        Mower.Moving = 1;              // thats north
        Mower.Energy = Evals;
// grow some more grass. Quite a small lawn (8x8)
        CreateLawn();
        Translate = TranslateROOT;
        while ( Mower.Energy )
        {
                Translate( ROOT );
// gets nought if it don't move and also stop infinite loops v.important
                if ( Mower.Energy == Evals ) return 0;
        }
        return ( 64 - CheckLawn() );
}
```

The major component of automatically defined functions comes in the translation of the root branch as shown below. If the genetic program chooses to translate the ADF0 branch the Translate function which is actually a pointer to a function is moved to point at the function TranslateADF0 then this branch is evaluated using the ADF0 macro (see the Macros section later in this report). Once this branch had been evaluated the returning vector is put in the global variable glbADF0 so the ADF1 branch can use it. The Translate function pointer is then shifted back to point at the TranslateROOT function and the glbADF0 is returned. The evaluation of ADF1 branch is similar expect this returns a value called vReturn.

```
FITNESS TranslateROOT( Gene *pg )
{
        switch ( pg->iValue )
        {
                case 1:
// ADF1 ->one args
                {
// note I send the variable to ADF1 through glbADF0
```

```
                      Vector vReturn;
                      Translate = TranslateADF1;
                      vReturn = Translate( ADF1 );
                      Translate = TranslateROOT;
                       return vReturn;
               }
               case 2:
               {
                      Translate = TranslateADF0;
                      glbADF0 = Translate( ADF0 );   // set up global variable...
                      Translate = TranslateROOT;
                      return glbADF0;
               }
               default:
               {
                      ExitSystem("TranslateMAIN" );
 // need to return something to stop warning though this will nerver be called
                      return glbADF0;
               }
        }
}
```

The TranslateADF0 function is interesting because it makes use of the real numbers attributes of gpc++. In initialising the random real function the range was set to 64. All real numbers as already stated reside after 32768 so the first process should be to get the actual value by simply subtracting 32768. A vector is then set up the first parameter being the modulus of the gene's value by 8 and the second the division of the value by 8. This gives a vector with the range 0 to 7 for each parameter.

```
FITNESS TranslateADF0( Gene *pg )
{
  switch ( pg->iValue )
  {
        case 1: return V8A( pg->pgChild );              // two args
        case 2: return PROG2( pg->pgChild );            // two args
        case 3: return LEFT();
        case 4: return MOW();
        default:
        {
//      and now the real number business, hope you like the way I have done it.....
               unsigned char ch = pg->iValue - 32768;  //set to begin at 0;
               Vector v_harvey;
               v_harvey.i = ch % 8;  // set the number of real numbers to 64 and then%8
               v_harvey.j = ch / 8;  //  / 8 to get the two sets of values.
              return v_harvey;
        }
  }
}
```

MACROS

When evaluating a genetic program the function should set it to a global variable genetic program (pgpGlobal). This allows the use of macros which make the decision upon which branch of the genetic program to evaluate a lot easier. The ADF1 macro, used in the lawnmower problem for instance, is equal to *(pgpGlobal->ppgHeader + 2) which is in English the pointer to the first gene within the third branch of the genetic program pgpGlobal. When calling this branch using macros it becomes simplistic just use Translate(ADF1). These macros can also be used to print out branches of the tree.

```
// macros of how to use ADF structure
#define ROOT *(pgpGlobal->ppgHeader)
#define ADF0 *(pgpGlobal->ppgHeader + 1)
#define ADF1 *(pgpGlobal->ppgHeader + 2)
```

ERRORS

There are commonly only three errors found in gpc++:

The allocation of memory reaches the limit and the system exits. This will tell you the function in which the error occurred.

The gp.ini file has not been set correctly and crashes the system. Unfortunately as the system stands safeguards against the wrong size of genetic program being used are impossible to implement.

The evolving code has the wrong number of functions and terminals defined in the translating procedures or in the creation of the function and terminal sets.

FUTURE WORK

Work on gpc++ has until now concentrated on developing and debugging the 'microkernel' of the genetic programming: the gene, gp and population classes and their definitions. On top of these generalised definitions a number of crossover, mutation or selection operators can be used altering the 'style' of the genetic programming run.

A major flaw is the difficulty in defining a particular problem. Each terminal and function set must be defined with unique values and a solution to a previous problem may be interpreted in a completely different way in a new problem. This could be produced by a very large switch ... case statement with each new problem adding further operators being added as a further case statement. Such a system will have intrinsic time delays due to comparing all the case statements and there will be further problems when different data types need to be returned.

One of the most interesting areas of artificial evolutionary research, in my humble opinion, is in multiple populations and co-evolutionary environments. As the system stands only one population is defined though I do give preliminary code in *eval.cc* into how multiple populations could be evaluated though the method described is very evaluation hungry.

As an ad addendum to this section here is a wish list for the future.

A syntactical measure for how similar two genetic programs are, which is also problem independent !
Demes which have their own particular genetic parameters.
Meta GP class which sits above the population judging convergence state of the system.

OTHER GENETIC PROGRAMMING IMPLEMENTATIONS

There are a number of implementations of genetic programming which can be found in the genetic programming archive (ftp://ftp.cc.utexas.edu /pub/genetic-programming/code) and a number of others which are in the hands of the more financially astute members of our fraternity. Each has its own advantages and disadvantages and this is not the place for a detailed analysis. So to mention just a few in order of chronology more than anything else;

LISP Genetic Programming by John Koza (hacked by James Rice, I presume)
SGPC Simple Genetic Programming in C by Walter Tackett
Stack Based GP by ?????
GPQuick by Andrew Singleton (detailed in BYTE Magazine Feb 1994)
GP in CLOS by Peter Dudey (A Common LISP implementation in 500 lines, very recent)
Smalltalk GP by ???????

ACKNOWLEDGEMENTS

The usual GP thank you's to John Koza for a truly wonderful architecture to be working within and James Rice, probably the greatest e-mail list moderator in the world.

Also I would like to thank Andrew Singleton (for how to do ADFs and the job offer, sort of), Jim Strauss (demes but no encapsulation sorry too many global variables), Eric Siegel (multiple populations done with demetic grouping ??), Patrick Ng and Nils Rognerund for their suggestions. Michael Timin also suggested the inclusion of the occasional comment within the code to help understanding. This took a very long weekend but has, hopefully, enhanced the code considerably. Thanks also to Howard Oakley and Bill Langdon who listened to my whinges at the recent AISB conference.

Personal acknowledgments go out to Vicki Harvey for putting up with a person who can only be considered as an opinionated 'pumpkin' head, Jon Rush a PhD partner of immense intellectual proportions and Rob Ghanea-Hercock, a nice research assistant with a wife.

A SMALL NOTE ON THE AUTHOR

Adam Fraser is currently (June 1994) moving into the third and final year of his PhD in evolving co-operative behaviours for autonomous agents. This, in no small way, scares the living daylights out of him. He has resided on this planet for 23 years and has continually upheld the belief that to take oneself seriously is the ultimate sin for which there in no penance. In July 95 the research will come to an end and he will be looking for a job which allows time for free thinking, an internet link and working until the early hours of the morning, your suggestions on this matter will be most welcome.

His work in genetic programming started about 2 years ago with a foray into the Santa Fe trail / artificial ant problem. Since then GP has become a major part of his work and as such the kernel code has been continually refined, improved and debugged. The version documented in this report has been programmed through numerous late nights (with Mark Radcliffe on Radio One) and caffeined early mornings, the comments in some blocks of code should definitely be read with this in mind.

He has numerous hobbies of the non-physical variety and does **NOT** want a nice house, a wife and 2.4 children. He also reads too much......

COMMENTS

If you wish to make comments on this document or the code then please feel free to e-mail me on the address shown on the front of the document. Please remember when making comments that this document and the code are work in progress and that they are also free. While this does not mean that my misdemeanours are any less it does mean that you should be gentle with me.

If you wish to send items such as research papers, books and food parcels (or even money ?!?) then please contact me using the address given below.

Adam Fraser,
Postgraduate Section,
Dept. Of Elec & Elec Eng.
Maxwell Building,
University Of Salford,
Salford,
M5 4WT,
United Kingdom

BIBLIOGRAPHY

Genetic programming is currently riding on the crest of a publishing wave, therefore rather than try to produce a bibliography of everything that is out in the ether I outline a few books and papers which should give the reader an excellent overview of the whole field. For a more detailed bibliography containing 152 genetic programming connected texts see the GP ftp site in /pub/genetic-programming/papers/152bib.txt which is John Koza's bibliography from the back of GP II referenced below.

'Genetic Programming', John R. Koza, MIT Press, 1992
'Genetic Programming II: Automatic Discovery of Reusable Programs', John R. Koza, 1994
'Advances in Genetic Programming', Kenneth E. Kinnear Jr. (Ed), 1994
'Genetic Programming For Feature Discovery and Image Discrimination', Walter Alden Tackett, *International Conference on Genetic Algorithms*, 1993
'Genetic Programming with C++', Andy Singleton, *BYTE Magazine*, February 1994
'Evolution of Corridor Following Behavior in a Noisy World'. *Simulation of Adaptive Behaviour*, August 1994

A SHORT AUTO-BIBLIOGRAPHY

Conference Papers

'Evolving Multiple Agent Behaviours For Biologically Inspired Robots', A.P.Fraser, J. R.Rush, D. P.Barnes, *Poster at Artificial Life IV*, July 1994
'Evolving Co-operation in Autonomous Robotic Systems', Rush, J.R., Fraser, A.P., Barnes, D.P., *IEE Int. Conf. Control,* 1994

Workshop Papers

'Putting INK into a BIRo: A Discussion of Problem Domain Knowledge for Evolutionary Robotics', Fraser A.P., Rush J.R., *Aritificial Intelligence and Simulation Of Behaviour Workshop on Evolutionary Computation*, 1994

'The Disposable BIRo: A Physical Representation of Evolutionary Robotics', Rush J.R., Fraser A.P., *Aritificial Intelligence and Simulation Of Behaviour Workshop on Models and Behaviours: Which way forward for robotics ?*, 1994

AND LAST BUT NEVER LEAST

The genetic programming mailing list is currently the most productive and useful lists I subscribe to. The experts (GP Wizards ??) keep their eye on foolish young babes (I include myself in this category) as they slowly move from crawling to walking in GP. If I was to explain to anyone what the utopian dream of academia was I would have to start with the example of the GP mailing list.

You can subscribe with an e-mail message to:

```
genetic-programming-request@cs.standford.edu
```

with the information below in the text message.

```
subscribe <Your Name>
```

The genetic programming archive contains code and papers and can be found at ftp.cc.utexas.edu in the directory /pub/genetic-programming. There is a FAQ available in this directory. Enjoy...

Le fin.

This document initially drafted 24 April 1994
This version was created on 11 June 1994
apf 94