DARWIN: A Genetic Algorithm Language

Arslan Arslan and Göktürk Üçoluk

Abstract This article describes the DARWIN Project, which is a Genetic Algorithm programming language and its C Cross-Compiler. The primary aim of this project is to facilitate experimentation of Genetic Algorithm solution representations, operators and parameters by requiring just a minimal set of definitions and automatically generating most of the program code. The syntax of the DARWIN language and an implementational overview of the the cross-compiler will be presented. It is assumed that the reader is familiar with Genetic Algorithms, Programming Languages and Compilers.

1 Introduction

The goal of the DARWIN Project is twofold - first, creating a programming language with support for Genetic Algorithm concepts and second, by requiring just a minimal set of specifications, generating all the necessary code automatically. The DARWIN Project thus involves a language design and its compiler implementation.

In general, if creation of a GA solution to a problem is desired, first the data representation is selected, the set of GA operators designed and the parameters specified. For example, lets decide that an array will store our chromosome and that we will implement 1-point cross-over with a binary flip mutation and a simple objective score function. Up to that point it is simple, but there is a lot to do. Next, we have to decide what kind of scaling and how selection will be performed. Thus, we create an array to store the statistics of the current population and define scaling and selection

A. Arslan (✉)
LOGO, Ankara, Turkey
e-mail: arslan.arslan@logo.com.tr

G. Üçoluk
Middle East Technical University, Ankara, Turkey
e-mail: ucoluk@ceng.metu.edu.tr

functions to use the information stored there-in. Finally, we code the genetic algorithm, to create a population and start evolving it until a termination condition is satisfied. At that point, we have a completely functioning Genetic Algorithm solution and we can start experimenting with different operators and GA parameters. This can be a time consuming process, since certain GA operators should be rewritten and modified, then the system should be tested against different inputs and parameters. It is custom to have hundreds of runs until a proper set of parameters is obtained. So a tool or a library is needed to facilitate creation of an initial system and afterwards help experimenting in the search of a better system.

Although GALib is easy to use, it is not that easy to extend. For example, creating a chromosome representation matching any of the available chromosome classes is easy, but when a more complex structure is needed a separate class should be derived and all the operators be redefined. This is still acceptable, but if creating a custom population object or even worse, a hand-made genetic algorithm object is required, then the programmer should be pretty much involved in the GALib object implementation and interaction details.

Another drawback of GALib comes from the fact that classes are as "general", i.e. data structure independent, as possible. So inefficient data storage arises, just because the implementation fails to specialize. To cover up, GALib provides many specialized implementations of the same object. For example, GAListGenome, GATreeGenome, GABinaryString and GAArray objects derive from the base class named GAGenome and provide support for specific genome representations. But this is not the end of the story, since the above mentioned four classes are in turn base classes for more and more specific data representations and this results in a fairly big object hierarchy.

In general Object Orientation is a handy concept, because implementation details can be hidden and inheritance is a fast way of complying to the overall library conventions, but expertise in C++ is required to accomplish tasks with even moderate difficulty. Another potential problem is the fact that inheritance hides the implementation details of the base class, so the resultant object inherits both the good and bad sides of his parents. As a result, the cumulative effect of the inefficient implementation during the path of derivation can add up to a point where performance is severely reduced.

The *General-purpose* systems go a step further, compared to a *Algorithm Library*, in the sense that they provide a high-level language to program their libraries. These high-level languages allow extremely fast prototyping, when experimenting with the system parameters. But, there is another reason for having such languages—the algorithm libraries are extremely complex and thus difficult to understand and extend in their low-level implementations. In addition, all such systems require a special *kernel* to execute these high-level GA instructions. The kernel may not be portable and can even rely on some parallel hardware.

As a solution, a programming language naturally recognizing and providing support for genome representations and their operators can be designed and implemented. By having a separate language and a compiler, all the necessary code can be generated for the unspecified genetic operators, thus giving a change for the developer to focus just on genome representation, operators and proper parameter set design. In addition, since genome data representation is known by the compiler, efficient code for dealing with the genome data structure can be automatically generated, thus eliminating inefficiency issues.

First, previous work related to the DARWIN project will be presented. In Sect. 4, an example of a DARWIN program will be given. In Sect. 5, An overview of the design of DARWIN and its implementation will be covered. We conclude in Sect. 7.

2 Related Previous Work

Genetic Algorithms received wide spread support due to the fact that they are robust and their complexity is not linear with respect to the number of parameters encoded in the solution. In fact, Genetic Algorithms were even used in obtaining near optimal solutions for many of the problems previously considered NP or NP-hard. But the task of producing a successful Genetic Algorithms system is not simple. This is due to the fact that experimentation with different GA operators and parameters is needed. So, some tools for facilitating GA program creation and experimentation with different genome structures and operators are needed.

In its most general case, the GA Programming Environments can be grouped into the following major classes [9]:

Application-oriented systems are programming environments hiding all the implementation details and are targeted at business professionals. These systems focus on a specific domain such as scheduling, telecommunications, finance, etc. A typical application-oriented environment contains a menu-driven interface giving access to parameterized implementations of genetic algorithms targeted at specific domains. The user can configure the application, change parameters and monitor program execution. Examples of application-oriented systems are EVOLVER, OMEGA, PC/BEABLE, XpertRule GenAsys, etc.

Algorithm-oriented systems are programming environments with support for specific genetic algorithm:

Algorithm-specific systems contain just a single powerful genetic algorithm. In general, these systems come in source code and allow the expert user to make changes. The implementations are usually modular in structure and there are nearly no user interfaces. Examples of these systems are ESCPADE, GAGA, GAUSCD, GENESIS and GENITOR.

Algorithm Libraries contain variety of genetic algorithms and operators are grouped in a library. These systems are modular, allowing the programmer to select among parameterized implementations of different algorithms and operators. Usually these libraries are implemented in C/C++ and Lisp. Examples of algorithm libraries are GALib, EM and OOGA.

Algorithm-oriented systems are often GNU Public Licensed and include free source code. Thus they can be easily incorporated into user applications.

Tool Kits are programming systems that contain many programming utilities, algorithms and genetic operators that can be used in a wide range of application domains. These programming systems subdivide into two:

Educational systems help the novice user to get familiar with Genetic Algorithm concepts. Typically these systems are very user friendly and support just a few options for configuring an algorithm. GA Workbench is the only example for an educational system.

General-purpose systems provide a comprehensive set of tools for programming any GA and application. These systems may even allow the expert user to customize any part of the software. Generally, these systems provide:

- Sophisticated user interface
- Parameterized algorithm library
- A high level language to program the system library
- Open architecture for expert user modification.

Examples include EnGENEer, GAME, MicroGA, PeGAsus and Splicer.

From the programmers point of view, *General-purpose* systems are the best tools available for programming GA systems. But, since these are not freely available, the current trend is creating libraries of functions or classes implementing predefined GA operators. Perhaps the most complete and easy to use Genetic Algorithms package is GALib. GALib is a C++ Library of Genetic Algorithm Components and is written and maintained by Matthew Wall, Mechanical Engineering Department of Massachusetts Institute of Technology [10].

GALib is a hierarchy of objects representing different GA constructs. The top level objects are GAGenome, GAPopulation and GAGeneticAlgorithm. The Genetic Algorithms operators are thus implemented as methods of these objects. The library features a full set of ready to use specialized versions of these top level objects, so a working program can be fastly prototyped, but experimentation usually requires modifications in the default behavior implemented by the classes. GALib provides two extension methods—deriving own classes to implement custom behavior, or defining new methods and instructing GALib to use them instead of the defaults.

3 The DARWIN Language

Stated briefly the solution of a problem using a Genetic Algorithm involves these important steps:

- Defining a representation
- Defining the objective score function
- Defining the genetic operators

The DARWIN language is designed to clearly distinguish Genetic Algorithm constructs—the genome and its operator set. It's language compiler has been designed to be a C cross-compiler. The target language is chosen to be C, since C is the most commonly used language with compilers present on the widest range of platforms. By generating standard C code, portability is ensured on the largest scale possible.

The DARWIN language is designed to be simple, GA oriented language. In order to achieve fast learning and adaptation, the syntax of the language resembles the standard C language syntax. For example, the DARWIN statements look just the same in C, with the minor difference that DARWIN is not as rich with build-in types as C and there are no pointers. The reason for excluding pointers from the language is that they are not currently needed, since all the storage in the generated code is linear. This does not lead to inefficient implementation since the generated code contains C pointers.

The DARWIN language, is capable of distinguishing *genetic algorithm constructs*. In the first place, DARWIN syntax provides means for defining *genes*, *chromosomes*, *populations* and the *genetic algorithm*.

The best possible way to represent them would be to have object classes. But since DARWIN generates plain C code, the DARWIN language has a special syntax construct called *moderators*. Moderator is a function that effectively associates an operation on a data structure, thus creating the effect of encapsulation present in Object Oriented programming. By introducing the concept of moderators, the DARWIN language becomes powerful enough to express a genetic algorithm construct and provide the standard set of operators associated with it. For example, the gene construct has the *print*, *initialize*, *crossover*, *mutate* and *evaluate* operations defined on it.

4 An Example of a DARWIN Program

4.1 Example Problem Definition and its Specifications

Consider an example problem of finding a two dimensional grid distribution of 20 given rectangles such that:

- No rectangles overlap.
- The *bounding box* (a smallest rectangle that includes all the rectangles) has a minimal area.
- Rectangles can be placed are allowed to be placed anywhere provided that one of their sides (any of it) is parallel to the horizontal direction. Furthermore all corners have to remain inside the grid (no clipping).
- The input of the problem consists of the dimensions of the rectangles.
- The grid dimensions are 256×256. The origin of the grid is labeled as (0, 0). This means that the upper-right corner of the grid is (255, 255). The horizontal axis is denoted with the letter x and the vertical axis with y. In a 2-tuple representation our convention will be writing (x, y) (as usual).
- Any coordinate (i, j) have positive integers for both i and j.
- Each rectangle is represented by a tuple $\langle Width, Height, x_{left}, y_{lower}\rangle$

4.2 DARWIN code of the solution

```
% a user defined function ;
external function int random(int start, int end);

gene TRectangle                  % start of genome
                                         definition;
{  int x;
   int y;
   frozen int width;
   frozen int height;
} < evaluator : RectangleEval, % evaluator moderator
                                  specification;
   init : RectangleInit;       % initializer
                            moderator specification;

chromosome TRectangles
{  TRectangle rectangles[20]; };

population Pop
{  TRectangles individuals[100]; };

algorithm GA
{ Pop population; };             % end of genome
                                       definitions;

% gene TRectangle evaluator moderator;
function float RectangleEval(TRectangle gene)
{
   if ((gene.x > 100) || (gene.y > 100))
     return 0;
   return 1;
}

% gene TRectangle initializer moderator;
function RectangleInit(TRectangle gene)
{
   gene.x = random(0,100);
   gene.y = random(0,100);
   gene.width  = random(0, 100-gene.x);
   gene.height = random(0, 100-gene.y);
}
```

5 Implementation

5.1 General Structure

The DARWIN project implementation consists of 4 major parts and an external code template database. It consists of

- Parser
- Post-processor
- Template Database
- Electra Engine
- Unparser

The general structure of the DARWIN cross-compiler is Fig. 1.

The *Parser* is the entity responsible for imposing the DARWIN Language syntax and converting a DARWIN program into a computer readable form—the parse tree. This parse tree is in turn fed into the *Post-Processor*, which in turn performs type-checking and verifies language semantics. The output of the Post-processor is an abstract syntax tree, equivalent to the parse tree but restructured in a form suitable for

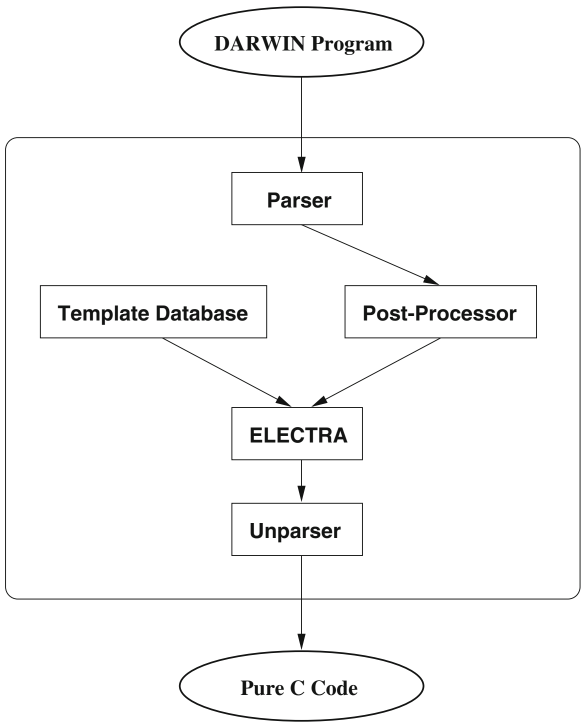

Fig. 1 Structure of DARWIN cross-compiler

traversal by the *Electra Engine*. In addition, the Post-Processor produces the symbol and type tables. Next, the abstract syntax tree is passed to the Electra Engine, which performs abstract tree translation and code generation. The Electra Engine relies on the presence of a *template database*, which is loaded prior to starting the engine. The template database is a text file containing abstract tree templates, which are matched inside the abstract tree of the DARWIN program and translated accordingly. The output of the Electra Engine is an abstract tree representing valid C code. The Unparser, takes this abstract tree and produces a formatted C program code.

5.2 Parser

The Parser converts the DARWIN syntax into its equivalent parse tree. The input is a DARWIN program file and the output of the Parser is a LISP structure, representing its parse tree. The Parser uses the parsing property list driven engine which is easily customizable and more efficient than the table-driven approach employed by LALR(1) parsing.

5.3 Post-Processor

The DARWIN Post-Processor operates in two stages. In the first stage it performs type-checking, manipulates the symbol table and collects all the necessary information that will be needed during the actual code generations. The Post-Processor is also responsible for tagging each branch in the abstract syntax tree. These tags will be used during the code generation phase, since the Electra engine will decide upon them.

After having performed the validity checks of the functions and genetic algorithm constructs, the Post-Processor enters its next stage, where the moderators specified in the genetic constructs are extracted from the *functions* section and are placed at the section they actually belong. During this phase, each of the genetic constructs are inspected and all of their moderators are searched in the functions section. Any inconsistency end up in an exception raise.

5.4 Electra and Template Database

Electra is a rule based unification pattern matcher/substituter and is the core code generating engine of the DARWIN compiler. It effectively traverses the input from the Post-Processor section by section, and decides when some additional code should be generated and how a given subtree should be transformed. So the Electra engine is traversing an abstract syntax tree, matching a given abstract syntax subtree and transforming it. The output is an abstract syntax tree, organized in sections.

6 Unparser

The Unparser is the abstract syntax to C code translator. The implementation adopts the one-pass pretty printer developed by Hearn and Norman [8]. The input of the Unparser is the output of the Electra Engine, where all of the syntactic sugars were removed, code translations done and any additional code generated. The output is a pretty printed C program, that corresponds to the user's DARWIN program.

6.1 Implementation Environment

The DARWIN Cross compiler is implemented in ALISP [4]. ALISP is an Algol like procedural syntactic sugar of Standard LISP [5]. Compared to LISP ALISP is much more readable. Technically they are identical, since an easy 1-to-1 mapping exists from ALISP to LISP counterpart. For portability reasons the project code is distributed as LISP code and ALISP is just used for ease of implementation.

The Standard LISP used belongs to Karabudak et al. [5]. The benefit of this LISP implementation is that it is free and contains an efficient LISP to C translator, thus achieving portability. In addition, when a LISP code is compiled a major speedup of up to 10 times of the interpretive execution speed can be achieved.

7 Conclusion

The DARWIN project aimed at creating a Genetic Algorithm programming to facilitate fast GA system creation and easy experimentation with different parameters and GA operators. We believe, that the goals are met.

Using DARWIN, a working system can be created with just presenting the genome definition and its evaluation function. In this case, the DARWIN compiler generates up to 85–90 % of the total code. Provided with a library of generator functions, a programmer can continue experimenting with different operator sets by just specifying library-provided moderators in the GA construct definitions. This still means, that the systems developer will be writing just 10 % of the total code and the rest will be automatically generated.

If the generator library does not provide the desired functionality, then the programmer will have to implement moderators using the DARWIN language. In an extreme case still we expect the DARWIN cross-compiler to generate around 45–50 % of the code on the average.

DARWIN together with LISP and ALISP is distributed freely under GNU Public License at the following URL: http://www.ceng.metu.edu.tr/ ucoluk/darwin/

References

1. Goldberg DE (1953) Genetic algorithms in search, optimization, and machine learning. Addison-Wesley Publishing Company, Inc, Reading
2. Bäck T (1996) Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press, New York, Oxford
3. Michalewicz Z (1992) Genetic algorithms + data structures = evolution programs. Springer, Berlin Heidelberg New York
4. Karabudak E, Üçoluk G (1982) ALISP—The manual
5. Karabudak E, Üçoluk G, Yılmaz T (1990) A C portable LISP interpreter and compiler, METU
6. Marti J, Hearn AC, Griss ML, Griss C (1979) The standard LISP report, SIGPLAN Notices, pp 48–68, no 10, ACM, New York, 14, 1979
7. Griss ML, Hearn AC (1981) A portable LISP compiler. Softw Pract Exp 11:541–605
8. Hearn AC, Norman AC (1979) A one-pass prettyprinter, SIGPLAN Notices, pp 50–58, no 12, ACM, New York, 14, 1979
9. Filho JR, Alippi C, Treleaven P (1994) Genetic algorithm programming environments. IEEE Comput J 27:28–43
10. Wall M (1996) GALib: A C++ library of genetic algorithm components, Ver.2.4