

A functional framework for the implementation of genetic algorithms: comparing Haskell and Standard ML (extended abstract)

Deryck F. Brown*, A. Beatriz Garmendia-Doval* and John A. W. McCall*

1. Introduction

We present our experience of developing a generic functional framework for the implementation of genetic algorithms (GAs). We have implemented two versions of the framework, one in Haskell and the other in Standard ML.

We have used this framework to implement a number of exemplar GAs from the literature, and an original GA to solve a problem in oil-well exploration. Our framework provides precise control over the stochastic processes underlying the GA, which allows for experimentation. The time taken to implement a GA using our framework is significantly reduced. As well as providing a generic implementation for much of the GA, it is also easy to re-use code from other GAs.

We have also used our framework to implement abstract interpretations of two GAs. In our framework, defining an abstract interpretation is equivalent to implementing a GA with non-standard genetic operators. We have used these abstract interpretations to study the abstract properties of the one-point crossover operator used in these GAs.

Developing our framework has given us insight into the problems of implementing a flexible, yet efficient, system in a functional language. So far, we have been happy with the flexibility of both the Haskell and the Standard ML implementations. We have experienced problems, however, with using the Haskell version to implement some (large) GAs. We have found it difficult to identify the cause of these problems due to the current lack of tool support.

From our experience, we give some performance figures for Haskell and Standard ML for several genetic algorithms. So far, we have found the Standard ML version to out-perform the Haskell version by about an order of magnitude. Part of this, however, is due to the difficulty in identifying the performance bottlenecks in the Haskell version.

2. Genetic algorithms

A genetic algorithm [1] is a heuristic search technique based on Darwinian ‘survival of the fittest’ evolution. To date, genetic algorithms have been widely used to solve problems in Science, Engineering, Medicine, Industry and Management.

To use a GA, the possible solutions to a problem must be represented as *chromosomes*. There must also be an *evaluation function* that maps chromosomes to their fitness values. The fitness value determines the quality of a solution; usually higher fitness values represent better solutions. A GA maintains a population of chromosomes and ‘breeds’ them, by using certain genetic operators, to produce new chromosomes. These offspring are then returned to the population. This process continues until some stopping condition is reached, e.g., after a certain number of iterations.

The implementation of a GA involves a large number of choices: the representation of solutions as chromosomes, the population size, the choice of genetic operators, and the rates affecting individual operators. How these choices combine to affect the overall performance of the GA is not well-understood, although there has been recent promising work in this area [2,3].

We aim to develop a better understanding of the performance of a GA by using abstract interpretation to identify more general properties. So far, we have implemented abstract interpretations of two GAs. Here we use only the fitness values of the chromosomes, and implement abstract genetic operators that work over fitness values, rather than on the chromosomes themselves. Our experiments demonstrate that, for these problems, the abstract interpretation produces results that are consistent with their standard interpretations [4].

3. Generic framework

The structure of a GA leads naturally to a functional approach to its definition. Each process step becomes a function to be defined. The GA itself is a function which takes a starting population and some random seeds, and outputs a sequence of successive populations.

* School of Computer and Mathematical Sciences, The Robert Gordon University, St Andrew Street, Aberdeen AB25 1HG, Scotland, UK. Email: {db,bgd,jm}@scms.rgu.ac.uk.

In both versions of the framework, we have used the module system of the language to decompose the problem into some generic modules, and some user-defined modules.

The generic modules shared by all GAs are:

- **Genetic algorithm.** This provides the common implementation of a genetic algorithm. It handles iteratively generating populations, including the tasks of selecting of parents and the breeding of offspring.
- **Genetic operators library.** This provides generic implementations of the most commonly used genetic operators for selection, crossover and mutation.
- **Random number generator.** This provides a system and language independent source of random numbers. It is an implementation of the Mersenne-Twister random number generator.
- **Probability.** This organizes the various stochastic processes used in the GA. It uses the random number generator to provide six independent sources of values such as probabilities, crossover positions, and mutation positions.

The user-defined modules are:

- **Environment.** This provides the definitions of the various GA parameters, e.g., population size, crossover rate, and mutation rate.
- **Chromosome.** This provides the definition of chromosomes, i.e., the representation of solutions. It includes the evaluation function for the problem.
- **Population.** This provides the definitions of the various genetic operators. Often these are simply taken from the genetic operators library module, but non-standard operators may be defined here.
- **Main.** This is the main program. It creates the initial population, and determines the stopping criterion. It uses the genetic algorithm module to perform the iterative evolution of the population.

In practice, the burden of defining a new problem is eased by providing templates for the user-defined modules. For example, a template chromosome module is provided for a GA that uses a bit-string representation for chromosomes.

Our framework allows a wide variety of GAs to be implemented, while at the same time standardizing their configuration. This makes possible abstract reasoning with wide applicability. Another practical advantage is that GAs can be implemented with minimal writing of new code.

4. Comparison

The Haskell implementation uses a monadic style to encapsulate the state of the stochastic processes and the GA parameters. To improve performance, its random number generator is written in C, and called from Haskell. The SML version uses imperative features to achieve this same encapsulation. Its random number generator is expressed in ML.

The Haskell version uses lists extensively: a population is a list of chromosomes, and a chromosome is a list of alleles (typically boolean values). The SML version uses vectors extensively: a population is a vector of chromosomes, and a chromosome is a vector of alleles.

The SML version has been profiled using the New Jersey SML compiler, and various code improvements performed. Where possible, these improvements have been applied to the Haskell version. No direct profiling of the Haskell implementation has been possible, due to difficulties with the compilers available at this time.

Apart from these differences, the implementations are largely equivalent. In fact, the SML version was developed as a translation of the Haskell version.

Our experience to date is that the Haskell implementation (compiled using GHC v4.06) is much slower than the SML version (compiled using SML of NJ v110). Our estimate is that it is approximately one order of magnitude slower when executed on a Sun E250 server. Memory consumption is also significantly greater in the Haskell version. We hope to obtain more accurate figures once a new version of GHC with profiling is released.

For one of our exemplar GAs, the Royal Road Problem, the Haskell version is rendered unusable by its memory requirements and time performance. This GA involves runs with varying parameter settings, some with a very large population, and some for a large number of iterations. These problems do not occur with the SML version.

5. Conclusion

Using a functional language to implement our generic framework has provided significant savings in development time and provided a highly flexible system.

The Haskell implementation is outperformed by the SML implementation, but it is disadvantaged by the current lack of tool support to analyze and improve the implementation.

Bibliography

- [1] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison Wesley, 1989.
- [2] E. van Nimwegen, J. P. Crutchfield, and M. Mitchell. Statistical dynamics of the Royal Road genetic algorithm. *Theoretical Computer Science*, 229:41–102, 1999.
- [3] M. D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. The MIT Press, Cambridge Massachusetts, USA, 1999.
- [4] D. F. Brown, A. B. Garmendia-Doval, and J. A. W. McCall. An abstraction interpretation of the Simple Genetic Algorithm using fitness. Submitted to *Foundations of Genetic Algorithms (FOGA '2000)*. Morgan Kaufmann, USA, 2000.