Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Adam Nohejl

Grammatical Evolution

Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc.
Study programme: Computer Science, General Computer Science

2009, revised 2010 (version 1.0.1)

Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Adam Nohejl

Grammatical Evolution

Kabinet software a výuky informatiky

Vedoucí bakalářské práce: RNDr. František Mráz, CSc.
Studijní program: Informatika, Obecná informatika

2009, upraveno 2010 (verze 1.0.1)

I thank to František Mráz, who supervised my work on this thesis, for his guidance and careful reading.

I also acknowledge the work done by others on the libGE and GEVA open-source projects. While my work does not derive directly from theirs, both projects helped me tremendously in designing AGE. I hope that I have succeeded in emulating what they do well, and that I have not repeated too many of their mistakes. I have certainly made enough mistakes of my own.

Last, but not least, this thesis would not be possible without the research and publications on evolutionary algorithms I have built on. Two works deserve a special mention: *Grammatical Evolution* by Michael O'Neill and Conor Ryan, who developed the methods of grammatical evolution, which are the subject of this thesis; and *A Field Guide to Genetic Programming* by Ricardo Poli and others, who kindly published an invaluable guide to GP as a free download.

The text was typeset using the pdfTEX typesetter and the LATEX macro package in the Latin Modern fonts and Helvetica. The diagrams were drawn in the fabulous OmniGraffle software. Plotting was done using the R graphics package, which is part of the open-source R Project for Statistical Computing.

Contents

Title: Grammatical Evolution
Author: Adam Nohejl
Department: Department of Software and Computer Science Education
Supervisor: RNDr. František Mráz, CSc.
Supervisor's e-mail address: `mraz@ksvi.mff.cuni.cz`

Abstract: Grammatical evolution (GE) is a recent grammar-based approach to genetic programming that allows development of solutions in an arbitrary programming language. Its existing implementations lack documentation and do not provide reproducible results suitable for further analysis. This thesis summarises the methods of GE and the standard methods used in evolutionary algorithms, and reviews the existing implementations, foremost the only actively developed one, GEVA. A new comprehensive software framework for GE is designed and implemented based on this review. It is modular, well-documented, portable, and gives reproducible results. It has been tested in two benchmark applications, in which it showed competitive results and outperformed GEVA 10 to 29 times in computational time. It is also shown how to further improve the performance and results by using techniques unsupported by GEVA, including new modifications to the previously published methods of bit-level mutation and "sensible" initialisation. The thesis and the software together form a solid foundation for further experiments and research.

Keywords: grammatical evolution, genetic programming, evolutionary algorithms.

Název práce: Gramatická evoluce
Autor: Adam Nohejl
Katedra: Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. František Mráz, CSc.
E-mail vedoucího: `mraz@ksvi.mff.cuni.cz`

Abstrakt: Gramatická evoluce (GE) je nový přístup ke genetickému programování s užitím gramatiky, který umožňuje vývoj řešení v libovolném programovacím jazyce. Její existující implementace nemají dostatečnou dokumentaci a neposkytují reprodukovatelné výsledky vhodné pro další analýzu. Tato práce shrnuje metody GE a standardní metody užívané v evolučních algoritmech, a zkoumá existující implementace, především jedinou aktivně vyvíjenou, software GEVA. Na základě toho je navrženo a implementováno nové komplexní prostředí pro GE. Je modulární, dobře dokumentované, přenositelné, a poskytuje reprodukovatelné výsledky. Bylo testováno ve dvou standardních testovacích úlohách, v nichž dosáhlo srovnatelných výsledků a 10krát až 29krát lepšího výkonu než GEVA. Dále je předvedeno, jak ještě zlepšit výsledky a výkon pomocí technik nepodporovaných v GEVA, mimo jiné nových úprav již publikovaných metod bitové mutace a „citlivé“ inicializace („sensible“ initialisation). Tato práce a software tvoří dobrý základ pro další výzkum.

Klíčová slova: gramatická evoluce, genetické programování, evoluční algoritmy.

Chapter 1

Introduction

Evolutionary principles have been used to solve computational and other problems automatically for almost half a century.[1] In the 1960s *evolutionary programming* (EP) pioneered by Lawrence J. Fogel was used to develop finite state machines, employing mutation and fitness-based reproduction. Roughly at the same time Ingo Rechenberg and Hans-Paul Schwefel applied *evolutionary strategies* (ES) to hydrodynamic problems, at first evolving only one real-valued vector at a time. Later, in the seventies, *genetic algorithms* (GA) developed by John Holland, his students and colleagues have put emphasis on sexual reproduction and fitness-proportional selection. While GA evolved fixed-length binary individuals (solution candidates), in the subsequent *genetic programming* (GP) individuals were in the form of computer programs. The best known work on genetic programming is John Koza's book of the same name (1992), although earlier examples exist.[2] These methods are collectively called *evolutionary algorithms* (EA).

From then on many refinements to evolutionary algorithms were both proposed by research and implemented in software projects ranging from simple demonstration tools to full-fledged environments. One of these refinements, called *Grammatical Evolution* (GE), is particularly promising, as it adds flexibility to individual representation and evaluation without having any special requirements on the evolutionary algorithm itself. That makes it possible for grammatical evolution to profit from advancements in EA research (for instance in fitness scaling methods or replacement strategies). Being used primarily to evolve computer programs, grammatical evolution is a special case of genetic programming regardless of the specific EA it is combined with. Although GE and its application to benchmark problems have been described (O'Neill and Ryan, 2003), there is still a lack of good software tools and verifiable results.

I hope this thesis and the accompanying software project *AGE* (Algorithms

[1]Darwinian principles were probably used in problem solving (as opposed to the preceding attempts of biologists at computer simulations of evolution) for the first time by Lawrence J. Fogel who is sometimes described as the "father of evolutionary programming". According to the web page `http://www.natural-selection.com/Press/2007/pr02262007.htm` (*Dr. Lawrence J. Fogel, Inventor of Evolutionary Programming, Dies at 78*): "Beginning in 1960, [Fogel] devised evolutionary programming, a radical approach in artificial intelligence that simulated evolution on computers to literally evolve solutions to problems." Koza (1992) cites *Artificial Intelligence through Simulated Evolution* published in 1966 by Fogel et al.

[2]A good overview of the development of EP, ES, GA and GP has been written by Banzhaf et al. (1998, sec. 4.3).

for Grammatical Evolution) will help the research of grammatical evolution advance. It summarises basics of the relevant theory embodied in various principles, algorithms and techniques, and implements them in a versatile, well documented tool. Several example applications provide results, which are compared to those of other implementations. It is, however, not intended to deliver a complete overview of EA and GE theory or an ultimate implementation of all algorithms available. Throughout the text I have made references to relevant works covering topics that are out of scope of this text when I was aware of them. I have striven to make the software framework easily extensible and to document it well. The text and the software project together form a solid foundation for further experiments and research.

The text is organised in the following chapters:

- Chapter 2 introduces the techniques of grammatical evolution and the underlying algorithms of genetic programming, discusses GE's advantages over tree-based GP, and provides a walk-through example of fitness evaluation using GE.
- Chapter 3 examines the major existing implementations, their advantages and drawbacks. The only (publicly available) implementation of GE comparable in extent and documentation to AGE is GEVA. I therefore focus on it.
- Chapter 4 sets goals for the software project based on that examination. AGE is designed to consist of a modular framework, a command line tool, and both user and developer documentation. While controlled from the command line, AGE also provides means for graphical presentation and analysis of the results.
- Chapter 5 describes and explains the major decisions taken during the design and development of AGE.
- Chapters 6 and 7 contain the user and developer documentation.
- Chapter 8 presents several benchmark applications and analyses their results obtained from AGE.
- Chapter 9 discusses the achievements and suggests directions for further development.
- Chapter 10 lists the changes made to the original text of the thesis, which was defended in September 2009.

The documentation, including the documentation of the API, is part of the thesis because there are frequent references between the documentation and the rest of the text.

Chapter 2

Introduction to Grammatical Evolution

In this chapter we will lay a theoretical foundation for a review of existing implementations of grammatical evolution, and, even more importantly, for designing a new implementation:

- in Section 2.1, we will explain the basics of genetic programming;
- in Section 2.2, we will tackle the concept of fitness, which is central to any evolutionary algorithm;
- in Section 2.3, we will learn about the distinction between genotype and phenotype, which is essential to grammatical evolution;
- in Section 2.4, we will describe grammatical evolution and show its advantages;
- in Section 2.5, we will go step by step through an example GE application;
- and finally, in Section 2.6, we will list and describe the building blocks of algorithms used in GP and GE.

2.1 Genetic programming

Genetic programming (GP) is a methodology for finding solutions to problems inspired by the evolutionary process and genetic mechanisms. GP distinguishes itself from other evolutionary algorithms by employing the form of a computer program for the solution candidates.[1]

The solution is being sought iteratively until a particular number of iterations, called *generations* in this context, is reached or good enough results are obtained. This process is called *evolution*.

Each generation consists of a fixed number of candidate solutions, a *population*

[1]Many of the principles described in this section apply to other EAs as well. I will again refer those interested in more general information to Banzhaf et al. (1998, ch. 4).

of *individuals*. Individuals are usually represented by trees or linear sequences;[2] either way they encode programs. Before the iterative process can begin, the population of generation zero has to be *initialised*. There are several ways to perform initialisation, some of which are suitable only for specific individual representations; they basically consist of assigning random data to individuals. Then a series of steps in each generation is performed:

Fitness evaluation: Programs represented by individuals, if syntactically correct, are executed and their fitness (a numerical value) is computed from their output. The function that assigns fitness values to the output, and consequently to the individuals, is called *fitness function* or *objective function* (a more general term used in optimisation). The fitness function depends only on the application. The fitness function does not therefore have to "know" anything about genetic programming. If individuals representing a syntactically invalid program occur, they are usually assigned the worst fitness possible. Low values may indicate either good or bad fitness. A fitness function is therefore associated with a specific ordering (preference for lower or higher values) and a range of output values.

Fitness scaling: For some methods of selection (see below) it is more convenient to have low fitness values indicate bad fitness, to limit the values to a certain range, or to ensure other properties. In that case fitness values need to be transformed from *raw* to *scaled*. The choice of a scaling method must respect both the ordering and the range of the fitness function values and the ordering and the range of fitness values supported by the selection scheme.

Selection: The selection of individuals for the next generation is designed to mirror natural selection: fitter individuals have a higher chance of survival and breeding. There are several distinct schemes, which combine random and fitness-based selection (the fitter the individual the more likely it is to be selected).

The selection scheme together with the scaling is said to determine the *selection pressure*: how strongly the system favours the fitter compared to the less fit. The implications of scaling and selection will be discussed in more depth in Section 2.2.

Application of genetic operators: The most important genetic operators are *crossover* and *mutation*. They correspond to their equivalents in genetics. Their specific realisation stems from the representation of individuals; too

[2]In tree-based GP the trees are constructed from *primitives*: the internal nodes from *functions*, which correspond to actual functions, operators or control structures, and the leaves from *terminals*, which correspond to variables, constants or functions without arguments (Poli et al., 2008, ch. 3). The use of linear sequences is actually common to several different approaches, most significantly: (1) efficient representation of some kinds of trees, in which case it is a special case of tree-based GP (Poli et al., 2008, sec. 2.1); (2) linear GP, which uses sequences of machine code or virtual machine instructions; (3) grammatical evolution, which is described in the next section. The most important ways of individual representation in GP and their variants are overviewed by Poli et al. (2008, sec. 2.1, ch. 6 (tree-based), ch. 7 (linear and graph-based)).

simply designed operators could easily render the program syntactically invalid. The operators are applied randomly, based on the given probabilities, on the selected individuals. Crossover, alternatively called *recombination*, combines two individuals by exchanging their parts, and produces one or two offspring. Mutation modifies a single individual, often by replacing a small part of it by random data or by negating a bit. Operators may either be mutually exclusive, or independent, mutation may, or may not, be limited to one application per individual. In case the crossover does not occur, individuals are simply copied over to the new generation.[3] When designing the operators, one of the aims is to produce valid programs.

The whole process of evolution in GP can be described with the following pseudocode for:
– n generations,
– p individuals in a population,
– R, a random number generator,
– p_x, p_m, probabilities of crossover and mutation.

```
EVOLUTION($n, p, R, p_x, p_m$)
 1  $P_0 \leftarrow$ INITIALISE-POPULATION($R, p$)
                                       ▷ array of individuals indexed $0 \,..\, p-1$
 2  for $i \leftarrow 0$ to $n-1$                ▷ or until a fit enough individual is found
 3      do $F_i \leftarrow$ EVALUATE-FITNESS($P_i$)
                                       ▷ array of fitness values of all individuals
 4         $F'_i \leftarrow$ SCALE-FITNESS($F_i$)      ▷ array of scaled fitness values
 5         $P_{i+1} \leftarrow$ empty array
 6         for $j \leftarrow 0$ to $p/2-1$
 7             do $I \leftarrow$ SELECT($R, F'_i, P_i$)
 8                $J \leftarrow$ SELECT($R, F'_i, P_i$)
 9                $\Big(P_{i+1}[2j+0],\ P_{i+1}[2j+1]\Big) \leftarrow$ CROSSOVER($R, p_x, I, J$)
10         for $j \leftarrow 0$ to $p-1$
11             do MUTATION($R, p_m, P_i[j]$)
                                       ▷ and possibly apply other operators
12  return the fittest individual based on $F$ and $P$
```

This code is very close to how actual implementations of core algorithms for GP look like. Variations of this algorithm employ additional operators, apply them in a different order, or regard them as mutually exclusive. Algorithms with more significant differences have also been designed, and are referred to with special names such as *steady-state* algorithm, as opposed to *simple* or *generational* (see Section 2.6.6).

As the methods of genetic programming are largely independent of the application, several libraries and environments for general genetic programming have been developed. They usually support several different evolutionary algorithms without any clear separation. To use them for a particular purpose, one has

[3]This default operator is sometimes confusingly called *reproduction* (foremost in Koza, 1992). Because it may suggest sexual reproduction, its exact opposite, I intentionally avoided this name.

to set up a specific algorithm and adjust its parameters, among others operator probabilities, and if needed, the specific way operators are performed, the mechanisms and parameters of selection, size of the individuals, size of the population, number of generations or the desired value of fitness, and most importantly, the fitness function. The function Evaluate-Fitness in the above pseudocode will call the fitness function for each individual and transform (scale) the values if needed. When implementing an application, one also has to design the structure of individuals and decide how they should be translated into a program.

There is no single most appropriate way to perform selection, the operators or initialisation of individuals, and there are no universal values for the quantitative parameters. Different applications can benefit from different approaches. It would also be misleading to expect every technique to be derived from genetic processes in nature, which are not yet fully understood and do not translate directly to the computer environment. While nature can inspire elegant algorithms, as we will see later, it is usually better to think of the methods used in GP more pragmatically as heuristics.

Genetic programming is described in more details and focus on practical implications by Poli et al. (2008).

2.2 Fitness, a broader view

Although the notion of a fitness measure is not peculiar to genetic programming, much less to grammatical evolution, a good understanding of it is essential both for implementing any evolutionary algorithm and for using it efficiently in an application. The fitness plays a pivotal role in evolutionary algorithms, it is a link between a particular application and the algorithm.

We have said that fitness values are numerical values from a certain range, associated with an ordering. Most general GP literature jumps directly from the natural concept of fitness to the concept of fitness as a numerical output of a fitness function (Koza, 1992; Banzhaf et al., 1998; O'Neill and Ryan, 2003; Poli et al., 2008). The distinction between the two, which is important to realise, is best worded by Goldberg (1989, p. 76):

> All this monkeying about with objective functions should arouse suspicion about the underlying relationship between objective functions and fitness functions.[4] In nature, fitness (the number of offspring that survive to reproduction) is a tautology. Large numbers of offspring survive because they are fit, and they are fit because large numbers of offspring survive. Survival in natural populations is the ultimate and only taskmaster of any import. By contradistinction, in genetic algorithm work we have the opportunity and perhaps the duty to regulate the level of competition among members of the population to achieve the interim and ultimate algorithm performance we desire.

[4]For Goldberg, *objective function* means raw, unscaled fitness function, while *fitness function* means scaled fitness function. This may be a more meaningful choice of terminology than ours, but uncommon in genetic programming.

This is precisely what we do when we perform scaling.[5]

It is indeed better to think of the raw fitness as of an objective function that has certain properties:

- It should be *continuous* (also used in this sense by Banzhaf et al. (1998)), which means that it should provide a gradient leading to the right solution. One should avoid fitness functions based on binary conditions, which leave "gaps" between values (as noted by Poli et al., 2008, sec. 13.8). When implementing the fitness function, precision is much less important than continuity.
- Its output values have a particular *ordering*: In *standard ordering* higher values are better, in *reverse ordering* lower values are better.[6]
- Its output values are from a particular *range*: It may have both the upper and lower bound, one of them, or none: A fitness function based on an error would have a range of $[0, m)$ (where m is the maximum error possible) or $[0, \infty)$, a fitness function for trading algorithms, allowing for both profit and loss, would be unbounded.[7]
- There is an optimal mapping from the output values to fitness in the sense of survival rate or selection probability.

Controlling and knowing these properties is vital to the construction of an efficient algorithm with appropriate methods for scaling and selection. The first property, continuity, is one to ensure, otherwise any evolutionary algorithm will perform poorly. The next two properties, ordering and range, narrow our choice of scaling and selection methods. The last one, the optimal mapping to selection probability, poses the most difficulty. We typically do not know this optimal mapping, but we should at least be aware that not all mappings are equal and be prepared to experiment with them:

- Suppose that we compute our fitness as a value combined (using a sum or otherwise) from several partial results. These are usually called *fitness cases*. What weight should be assigned to each case? Additionally, if the cases are errors of the candidate solution at several points, should we combine their absolute values or their squares? These transformations already take place in the fitness function, but they contribute to the mapping in the same way as scaling does. (To square the errors is, in effect, to apply a power law scaling to the fitness cases; see Section 2.6, p. 29.) It is also worth note that a transformation applied to fitness cases, in contrast to the final fitness value, can even change the relative order of fitness values.

[5]Goldberg was only concerned with various methods of fitness-proportional selection (see footnote 13, p. 28), he did not consider the now mainstream variant of tournament selection, which ignores scaling. I venture to say that if he did, he would speak of "scaling *and selection*".

[6]I have chosen these two names to avoid using domain-specific classification, such as "profit" or "error", in general discussion. The terms maximisation and minimisation, while useful, apply rather to the algorithm than to the fitness. The standard ordering is not to be confused with Koza's (1992) "standardised fitness", which is incidentally defined to have reverse ordering.

[7]Every fitness function, when implemented, is implicitly bounded by the maximum floating point value allowed by the implementation. Higher values are usually truncated to that maximum.

- Suppose that our raw fitness values have reverse ordering, while the selection method demands standard ordering. Mapping the values with simple inversion ($1/x$), and thus reordering them, is certainly possible, but chances are that it is not the optimal mapping. Compare its effect with Koza's adjusted fitness (Section 2.6, p. 29), which was already designed to magnify differences between values close to zero. Do we really want to exaggerate differences between very small values that much at the expense of differences in the rest of the spectrum?

- Suppose that we use tournament selection (Section 2.6, p. 28), a method described as "alternative" by Koza (1992), but which, according to Banzhaf et al. (1998) six years later, "has recently become a mainstream method for selection." They attribute that to the possibilities it opens for parallelisms, but Poli et al. (2008) highlight another important trait:

  > Note that tournament selection only looks at which program is better than another. It does not need to know how much better. This effectively automatically rescales fitness, so that the selection pressure on the population remains constant. [...] Since tournament selection is easy to implement and provides automatic fitness rescaling, it is commonly used in GP.

  Tournament selection can be used easily with any range and ordering of fitness values and it does automatically rescale the fitness, to be sure, but there is no guarantee that it does so in an optimal way. The selection pressure is a good guide for the choice of scaling and selection, but it is not the only measure of their quality. Note that both tournaments of different sizes and Goldberg's linear scaling (Section 2.6, p. 29) in conjunction with fitness-proportional selection are designed to maintain selection pressure. If they gave equivalent results in all applications, tournament selection with tournaments of size of 2 without any scaling would be universally adopted, as it is the least resource intensive one.

Scaling methods and selection schemes should be applied carefully. Results of the algorithm can often be improved by experimenting with them. One should not forget that to apply no transformation or an innocent-looking formula ($1/x$ or x^2) is also a choice of mapping from raw fitness to selection probability. This applies both to scaling and to transformation done inside the fitness function.

2.3 Genotype and phenotype

Until now we have assumed either that individuals are equivalent to programs they represent, or that they are somehow automatically translated before execution. In genetic programming this translation process is usually called simply *mapping*, or in reference to the analogous process in genetics, *genotype-phenotype mapping*. Genotype means the actual genetic information, while phenotype is defined by the characteristics observable from interaction with the environment. The individual representation format (*genotype*), the program language (*phenotype*), the operators, and the mapping have to be designed and implemented with respect to each other, and have to form a coherent whole. In genetic programming a tree-based representation is the most often used one (and traditionally mapped to Lisp).[8]

Tree-based representation naturally expresses syntax trees of programs. It is suitable chiefly for languages with automatic garbage collection and it may not be efficient enough due to increased demands on storage and memory management. In some applications the problems can be remedied by using GP functions with the same number and type of arguments and representing trees linearly. (See footnote 2 on page 14 for terminology. The efficiency issues are discussed by Poli et al. (2008, p. 10).) All this along with the desire to produce syntactically valid and executable programs also has to be taken into account when designing the operators.

From the practical point of view this approach places additional burden not only on the GP environment developer, but also on the application developers, who need to take special considerations when choosing a language used for the solutions, and then adjust the mapping and operators accordingly. The use of trees for individual representation also lacks a clear genotype-phenotype distinction, which is present in living organisms and is fundamental to the theory of genetics (see for instance Lewontin, 2008).

2.4 Grammatical evolution

Grammatical evolution (GE) is an approach that builds on genetic programming.[9] It facilitates the genotype-phenotype mapping of individuals, and consequently the design of their structure, regardless of application. It also brings other benefits "for free", such as validity of programs being independent of operator implementation.

This variant of genetic programming, which is supposedly closer to the natural process, has been described in *Grammatical Evolution* by Michael O'Neill and Conor Ryan (O'Neill and Ryan, 2003), which is based on O'Neill's earlier PhD thesis. As of this writing this is the most thorough work published on the subject. As the authors note (O'Neill and Ryan, 2003, p. 34), there have been several other attempts to use grammars with GP. In this case the word *grammatical* refers to the use of a formal grammar in the mapping process, more specifically a context-free grammar specified in Backus-Naur form (BNF).

The genotype is simply represented by a string of integer values. In analogy to the DNA helix and nucleobase triplets, the string is often called *chromosome*,

[8]The most important work on GP to use Lisp is John Koza's *Genetic Programming.* Koza (1992, sec. 4.3) gives several reasons for choosing Lisp are given, all of which derive either from the straightforward correspondence between program and data in Lisp and their parse trees, or from Lisp's relative ease of use and efficiency (on a Lisp machine) at the time. While the performance and ease of use reasons may not apply today, the correspondence between the parse tree and the program can still be a motivation to use Lisp for pure tree-based GP. Compare this to grammatical evolution described in the next section.

[9]This is also the stance of O'Neill and Ryan (2003). They, however, use the term *evolutionary automatic programming* to refer to GP: "We feel that the use of the term Evolutionary Automatic Programming, instead of GP with its various interpretations, is preferable [...]" I opted for the more widely used term *genetic programming* (Banzhaf et al., 1998; Poli et al., 2008).

and the values it consists of are called *codons*. Codons consist of a fixed number of bits. The mapping to phenotype, which can be seen as an analogy to the process of genetic translation, proceeds by deriving a string as outlined below in the pseudocode for Derivation-Using-Codons. The procedure accepts the following parameters:

– G, a context-free grammar in BNF. Note that BNF ensures that there is a non-empty ordered list of rewriting rules for each nonterminal.

– S, a nonterminal to start deriving from. The start nonterminal of G should be passed in the initial call.

– C, a string of codons to be mapped. Note that it is passed by reference and the recursive calls will sequentially consume its codons using the procedure Pop.

```
Derivation-Using-Codons($G, S, C$)
 1  $P \leftarrow$ array of rules for the nonterminal $S$ in $G$ indexed from zero
 2  $n \leftarrow$ *length*$[P]$
 3  **if** $n = 1$                          ▷ only one rule (no choice necessary)
 4     **then** $r \leftarrow 0$
 5  **elseif** *length*$[C] > 0$            ▷ choice necessary, enough codons
 6     **then** $r \leftarrow$ Pop$(C)$ mod $n$
 7  **else error** “Out of codons”          ▷ or *wrap* $C$, more on that later
 8  $s \leftarrow$ string of symbols at the right-hand side of the rule $P[r]$
 9  $t \leftarrow \lambda$
10  **foreach** $A \leftarrow$ symbols of $s$ sequentially
11        **do if** $A$ is terminal
12              **then** $t \leftarrow t\,.\,A$
13              **else** $t \leftarrow t\,.\,$Derivation-Using-Codons$(G, A, C)$
14  **return** $t$
```

This way we have achieved a convenient simplification. It is sufficient to describe valid programs in BNF without further specification of structure and interpretation of individuals. Operators on the chromosome strings can be implemented in a straightforward way completely independent of phenotype. However, several new problems seem to have arisen:

- Some of the individuals will still be invalid (when line 7 is reached.)
- As every codon is interpreted modulo n, where n is the number of rules for the current nonterminal (see line 6), and it is interpreted at most once, the information contained in a codon is never fully used.
- Additionally, the derivation is often completed before the codons are exhausted, which means that part of the genotype does not have any effect.

Grammatical evolution solves or justifies the supposed problems quite easily:

- When the end of the chromosome is reached, the process continues from its start, as if the chromosome was circular instead of linear. This process is dubbed *wrapping* (O’Neill and Ryan, 2003). Only a certain finite number of such wrappings is allowed. Nevertheless, the number of invalid individuals is reduced (O’Neill and Ryan, 2003, sec. 6.2).

- In addition to that, wrapping makes it possible for codons to be interpreted more than once with respect to different sets of rules using different parts of the information contained in a codon.
- The unused codons, as well as different parts of the information contained in the used ones, can actually come into effect in later generations after genetic operators are applied.

A supportive argument for wrapping and using different parts of the genome under different circumstances is their similarity to genetic phenomena: wrapping is similar to *gene overlapping*, and the latter two cases can be seen as instances of *genetic code degeneracy*. Wrapping can be controlled by changing the number of wrapping events allowed per mapping (it can be disabled entirely). Degeneracy can be controlled to some extent by changing the codon size (the minimum number of bits per codon is determined by the highest number of rules for a nonterminal in the grammar). O'Neill and Ryan (2003, p. 72) recommends 8-bit codons, and employs them in all examples, which have up to four rules for one nonterminal. They, however, “do not expect this codon size to be the optimal across all problems.”

More information, including parallels with biological processes and evidence that both wrapping and degeneracy can improve performance, can be found in (O'Neill and Ryan, 2003).

2.5 GE example: symbolic regression

Let's take look at how a GE fitness function may look like and how it would evaluate several example individuals for a specific problem: *symbolic regression*. Symbolic regression tries to find a function represented by an expression (thus being *symbolic*) to fit a given set of target function values at specified points (thus being a *regression*).

This is a canonical example of a GP problem because its fitness function can be implemented easily, for instance using a sum of absolute errors at the specified points, and it can provide a gradient leading to the right solution.

In a real-world setting the target values would be determined experimentally and the solution would be expected to match the values within a specified error range; fitness could also partially depend on the expression length to avoid overfitting.

For our example we will use target values computed from an actual polynomial $x^5 + x^4 + x^3 + x^2 + x$ at 20 points evenly distributed over the range of $[-1, 1]$.

More general information about symbolic regression and genetic programming can be found in (Koza, 1992).

2.5.1 Grammar

In the following examples we will compose grammars producing expressions with a C-like syntax, which are valid in several programming languages (C, Java, Lua and others). Pretending that we do not know that the target function involves only addition and multiplication, we could start with a grammar providing for all sorts of operations and functions (see Listing 2.1).

```
<expr>    ::=  <expr> <op> <expr>
          |    ( <expr> <op> <expr> )
          |    <pre-op> ( <expr> )
          |    <var>
<op>      ::=  +    | -    | /    | *
<pre-op>  ::=  sin  | cos  | exp  | log
<var>     ::=  x    | 1.0
```

Listing 2.1: Example grammar rules in BNF for symbolic regression (many operations and functions).

This specification of rules in BNF also implicitly defines nonterminal symbols `expr`, `op`, `pre-op`, `var`, and terminal symbols `+`, `-`, `/`, `*`, `sin`, ..., `1.0`. By convention the first nonterminal, in this case `expr`, is the start nonterminal. The sets of nonterminals, terminals, production rules, and the start nonterminal together define a grammar. As left-hand sides of BNF rules can only be nonterminals, it is a context-free grammar.

At this point we should note that this grammar was composed somewhat arbitrarily. We could experiment with different ways of constructing the expression (for instance by omitting the rule `<expr>` $\rightarrow$ `<expr> <op> <expr>`). We could choose different functions and operations based on what we expect in our problem space. Another important decision is whether to include the so-called *ephemeral constants*. In this case we have chosen to include only `1.0`, letting other constants evolve spontaneously. Instead we could either explicitly include more constants, or leave them out completely (`1.0` can evolve as `x/x`). (Dempsey et al., 2009 devote two chapters to evolution of constants in GE.)

The choice of grammar both delimits the search space (without exponentiation or the exp function we cannot find a function involving exponential dependency) and favours certain solutions (the grammar specified in Listing 2.1 gives arithmetic operators in `op` twice the probability of functions in `pre-op`, note the two occurrences of `<op>`).

Having realised how large the search space, and consequently the resources needed for the search, would be for a grammar with so many functions, we will first try our luck with a simpler grammar (see Listing 2.2).

```
<expr>  ::= ( <expr> <op> <expr> ) | <var>
<op>    ::= + | - | *
<var>   ::= x | 1.0
```

Listing 2.2: Example grammar rules in BNF for symbolic regression (simplified version).

2.5.2 Fitness function

Our fitness function will be computed as a sum of absolute errors (differences from the target values) at the given 20 points. As a result, zero fitness is the best possible, the larger the value the worse: the values are nonnegative with reverse ordering. We will let the evolution iterate until either a small enough fitness is reached[10] or a certain number of generation has evolved.

The implementation will involve evaluation of expressions. The grammar already guarantees that an expression is syntactically valid, but if we have chosen the first more complex grammar, the expression might fail at runtime because of division by zero or invalid argument for the `log` function, depending on how these conditions are dealt with in our programming language of choice. In any case it is necessary to handle the failures. Let's decide that we search for a function defined at all tested points, and thus assign the worst fitness possible if evaluation fails or the result is undefined.

Let's summarise the constants and procedures we will use in the fitness function:

– G is the context-free grammar in BNF as specified above.

– S is its start symbol (`expr`).

– X is an array of the 20 points $(-1.0, -1.1, \ldots, 1.9)$ where the function will be evaluated.

– Y is an array of the target $(x^5 + x^4 + x^3 + x^2 + x)$ values at points X.

– $f_{\max}$ is the maximum representable fitness value (the worst possible one).

– DERIVATION-USING-CODONS is based on the function defined on page 20 extended with the possibility of wrapping. It may fail with an error when the codon string needs to be wrapped too many times.

– EVALUATE-EXPRESSION(e, x) is a function that evaluates the expression string e (using a compiler, an interpreter or a virtual machine) at point x (that is with `x` $= x$). It may fail with an arithmetic error as described above.

```
SYMBOLIC-REGRESSION-FITNESS(individual)
 1  try
 2          $expr \leftarrow$ DERIVATION-USING-CODONS($G, S, codons[individual]$)
 3  catch "Too many wrappings"
 4          return $f_{\max}$
 5  $e \leftarrow 0$
 6  for $i \leftarrow 0$ to $length[X]$
 7      do try
 8              $e \leftarrow e + |$EVALUATE-EXPRESSION$(expr, X[i]) - Y[i]|$
 9          catch "Arithmetic error"
10              return $f_{\max}$
11  return $e$
```

This is a complete fitness function, which could be called from EVALUATE-FITNESS in our pseudocode for EVOLUTION on page 15.

[10]Expecting the fitness to reach zero exactly is not a reasonable requirement, even in this artificial example where the target values are precomputed. Due to imprecise floating point computations, two expressions that are mathematically equivalent in terms of value can give slightly different results. For instance $(x + x^3)(x + x^2) + x$ is certainly equivalent to $x^5 + x^4 + x^3 + x^2 + x$ but it might not give the same numerical result.

2.5.3 Evaluation of example individuals

Suppose that the following three individuals A, B, and C are passed to our fitness function by the evolution algorithm. Evaluation of each one will start by deriving the phenotype string using the codons of their chromosome. It is done according to the grammar defined in Listing 2.2 starting from the nonterminal `<expr>`.

- Individual A: $\mathit{codons}[A] = (10\ 11\ 99\ 62\ 33\ 22)$. Derivation proceeds as shown in Figure 2.1. Every codon is used exactly once. The result is a valid expression `1 * x`. The expression is evaluated as specified in the second part of Symbolic-Regression-Fitness, resulting in a fitness value of approximately 8.767.

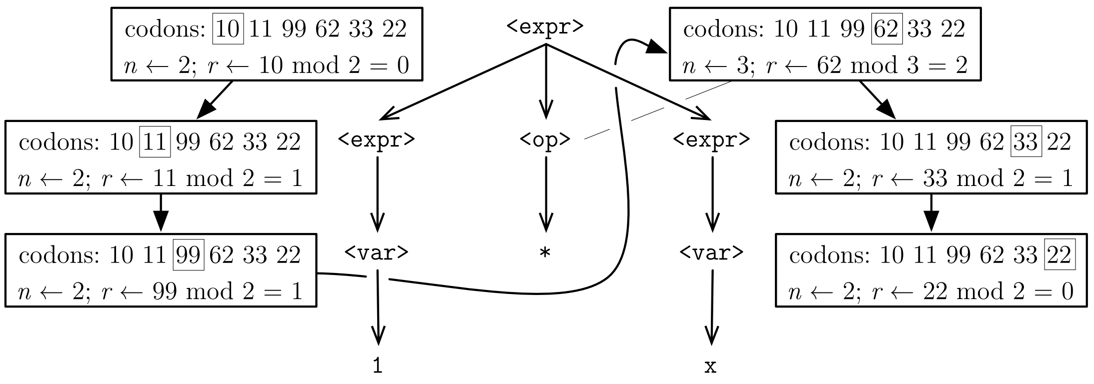

codons: 10 11 99 62 33 22
$n \leftarrow 2$; $r \leftarrow 10 \bmod 2 = 0$
codons: 10 11 99 62 33 22
$n \leftarrow 2$; $r \leftarrow 11 \bmod 2 = 1$
codons: 10 11 99 62 33 22
$n \leftarrow 2$; $r \leftarrow 99 \bmod 2 = 1$
<expr>
<expr>
<op>
<expr>
<var>
*
<var>
1
x
codons: 10 11 99 62 33 22
$n \leftarrow 3$; $r \leftarrow 62 \bmod 3 = 2$
codons: 10 11 99 62 33 22
$n \leftarrow 2$; $r \leftarrow 33 \bmod 2 = 1$
codons: 10 11 99 62 33 22
$n \leftarrow 2$; $r \leftarrow 22 \bmod 2 = 0$

Figure 2.1: Individual A: no wrapping occurs. (Terminals '(' and ')' left out for clarity.)

- Individual B: $\mathit{codons}[B] = (42\ 28\ 10)$. Derivation proceeds as shown in Figure 2.2. As every codon mod $2 = 0$, the same rule is applied over and over without reaching any terminals. After three applications of the rule `<expr>` $\rightarrow$ `( <expr> <op> <expr> )`, wrapping is applied, resulting in an infinite loop. Regardless of the maximum number of wrappings allowed, this individual is invalid, and it is assigned the worst possible fitness value, $f_{\max}$.

- Individual C: $\mathit{codons}[C] = (16\ 21\ 22\ 24\ 10\ 11\ 80\ 32\ 60\ 59\ 13)$. Derivation proceeds as shown in Figure 2.3. The chromosome is exhausted before terminals are reached in all branches of the derivation tree. After one wrapping the derivation is successfully finished. Note that the first codon is interpreted modulo 3 instead of modulo 2. The result is a valid expression `(x + (x * (1 - x) )`. The expression is evaluated as specified in the second part of Symbolic-Regression-Fitness, resulting in a fitness value of approximately 17.009.

The evaluated individuals have been assigned fitness values

$$f(A) \approx 8.767, \quad f(B) = f_{\max}, \quad f(C) \approx 17.009.$$

What does that mean for their chances of survival and reproduction? The exact answer of course depends on the fitness scaling and selection scheme, some of which will be discussed in the following section. A common scenario would be:

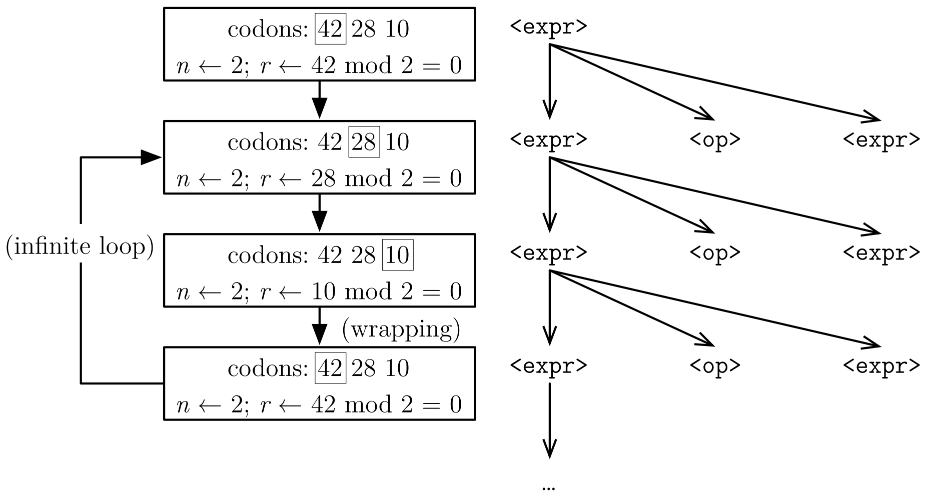

codons: 42 28 10
$n \leftarrow 2$; $r \leftarrow 42 \bmod 2 = 0$
codons: 42 28 10
$n \leftarrow 2$; $r \leftarrow 28 \bmod 2 = 0$
(infinite loop)
codons: 42 28 10
$n \leftarrow 2$; $r \leftarrow 10 \bmod 2 = 0$
(wrapping)
codons: 42 28 10
$n \leftarrow 2$; $r \leftarrow 42 \bmod 2 = 0$
<expr>
<expr> <op> <expr>
<expr> <op> <expr>
<expr> <op> <expr>
...

Figure 2.2: Individual B: wrapping results in an infinite loop.

– roulette-wheel selection, which selects individual x with probability proportional to its fitness $f'(x)$,

– Koza's adjusted fitness scaling $f'(x) = 1/(1 + f(x))$, which can be used to transform our error-based fitness values to values suitable for the roulette-wheel selection.

In that case the probabilities of selection would be proportional to scaled fitness values:

$$\begin{aligned}
f'(A) &= 1/(1 + f(A)) \approx 1/(1 + 8.767) \approx 0.102,\\
f'(B) &= 1/(1 + f(B)) = 1/(1 + f_{\max}) \approx 0.000,\\
f'(C) &= 1/(1 + f(C)) \approx 1/(1 + 17.009) \approx 0.056.
\end{aligned}$$

Thus the individual A would have about twice the chance of individual C to survive and breed, while the invalid individual B would almost never be selected.

2.6 Search algorithm elements

Section 2.1 presented a basic search algorithm called simple or generational. This algorithm was based on smaller building blocks: initialisation, fitness evaluation, selection, and operators, which can be implemented in various ways. Additionally even the algorithm itself can be altered, usually to form new generations in a different way, or not to use generations at all.

As follows from the above description of grammatical evolution, any implementation that can operate on variable-length integer strings can be used for GE. It is even possible to apply GE to integer vectors, that is fixed-length integer strings. This, however, would impair the algorithm's ability to evolve a

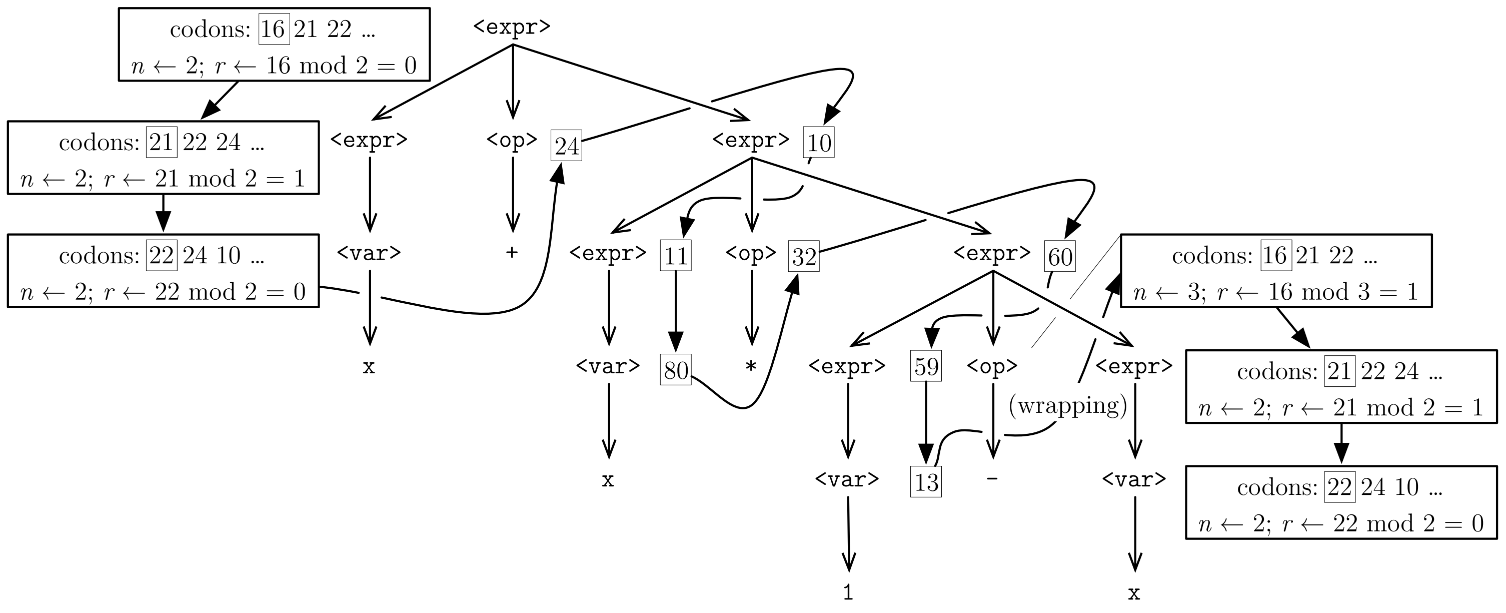


Figure 2.3: Individual C: wrapping results in reinterpretation. (Terminals ‘`(`’ and ‘`)`’ left out for clarity.)

suitable string length on its own.[11] The following paragraphs describe several variants of the algorithm elements commonly used for GE.

2.6.1 Initialisation

Two methods to create the initial population exist:

Random initialisation produces a population of individuals with chromosome lengths evenly distributed over a certain range and chromosomes filled with random values. This very simple method does not take into account any properties of the grammar used for evaluation. Random initialisation can also be used for fixed-length genotype, in which case the chromosome length is constant.

Sensible initialisation (described in more details by O'Neill and Ryan, 2003, sec. 8.8), on the other hand, employs the grammar that is used for the mapping process to produce individuals with certain properties. It is modelled after the *ramped half-and-half* technique devised by Koza for tree-based GP (Koza, 1992, sec. 6.2, app. C) and its aim is to produce a wide variety of derivation tree shapes and sizes.

2.6.2 Fitness evaluation

Fitness evaluation is entirely application dependent, except of course for the use of genotype-phenotype mapping with a grammar. Section 2.5 gives an example of fitness evaluation in GE.

2.6.3 Selection

All mainstream methods of selection are based solely on the fitness values, any of them are therefore applicable to grammatical evolution. We will describe the most common two of them:

Roulette-wheel selection allocates each individual a section of a hypothetical roulette wheel proportional to its fitness. An individual is selected by spinning the wheel.[12]

When the process is repeated, a previously selected individual can be selected again, it is, therefore, a selection *with replacement.* Fitness values must be non-negative, finite and high values must indicate high fitness. For this method, not only the relative order, but also the relative magnitude of the fitness values is very important. For instance fitness values constrained

[11]In particular, if the sought solution involved more applications of rules for nonterminals then there were codons in the vector, wrapping would be the only means to find that solution. As shown in an experiment in (O'Neill and Ryan, 2003), wrapping is valuable only for some applications, and if it is not, it can cause the chromosome length to actually increase even more (O'Neill and Ryan, 2003, sec. 6.2). Therefore, it should not be relied on to produce phenotype of variable complexity from fixed-length genotype.

[12]If we assume individuals numbered from 1 to n, and a distribution defined by a probability mass function $f(i) =$ [fitness of individual i], then the roulette-wheel selection can equivalently be defined as generation of a random variate of that distribution.

to a small range with regard to their average value, will result in virtually the same probabilities of being selected, and are therefore unsuitable for this method. Alternative name for this method is *stochastic sampling with replacement* (Goldberg, 1989, Goldberg himself, however, prefers to call it roulette wheel).[13]

Tournament selection picks a random group of t individuals from the population, and then selects the best one of the group. Unless stated otherwise the method is usually understood as being *without replacement* within a given tournament. This ensures that there are t distinct competitors as corresponds to the natural notion of a tournament (Koza, 1992; Banzhaf et al., 1998).[14] Many variants of tournaments exist, but there is no fixed terminology for them: being "with replacement" may indicate replacement within a tournament or among tournaments, being "stochastic" may apply to selection of the competitors or to the selection of the winner of tournament. Unless the competitors for the tournament are themselves selected with a more elaborate method, the exact values of fitness do not matter for tournament selection, only how they compare to each other within the tournament.

2.6.4 Fitness scaling

Although fitness scaling occurs before selection, we discuss the methods in reverse order to reflect that fitness scaling is an optional complement to selection and should be always chosen in conjunction with a specific selection algorithm. When one or more scaling transformations are applied, the selection is based on the final scaled values.

Any fitness scaling methods can be used with GE, the only notable exception that would require modifications is fitness sharing:

Reversal, the simplest transformation, is used to reverse the ordering by subtracting the raw fitness value from a constant:

$$f' = \begin{cases} c - f & \text{if } f \geq C \\ 0 & \text{otherwise} \end{cases}$$

The constant c is usually chosen so that $f \geq c$, a commonly used value is the maximum fitness in the current generation. This scaling is most useful in two cases:

– fitness-proportional selection is being used, but our fitness values represent cost, distance, or error rather than profit or utility (Goldberg, 1989);

[13]While some authors (Koza, 1992; Banzhaf et al., 1998) also use the term *fitness-proportional selection* for roulette-wheel selection, I reserve it to its general meaning: any selection method that is fitness-proportional. Goldberg (1989, p. 121) lists five of them, and adds a variant of tournament (see below) that uses a fitness-proportional method for picking competitors.

[14]Surprisingly Koza's own sample code from the cited book picks the competitors with replacement. This seems to be more common in implementations, probably due to the natural laziness of programmers.

– our fitness values are in the correct order, but we want to apply a different scaling that changes the order.

Koza's adjusted fitness is designed to "exaggerate the importance of small differences in the [input] value as [it] approaches 0 (as often occurs on later generations of a run)" (Koza, 1992, sec. 6.3.3). It applies the following formula to nonnegative values with reverse ordering to get a nonnegative values with standard ordering:

$$f' = \frac{1}{1+f}$$

Koza uses this transformation throughout his book. If necessary a reversal, as defined above, is applied beforehand to change the ordering.

Goldberg's linear scaling with factor k is a simple linear formula with coefficients a and b such that using roulette wheel selection, an average individual gets one expected copy in the new population and the best individual gets k copies:

$$f' = af + b$$

Computation of the coefficients and other details are described in Goldberg's *Genetic Algorithms* (Goldberg, 1989, ch. 3: Fitness Scaling, ch 4: Scaling Mechanisms). Suggested values for k are 1.2 to 2 for populations of 50 to 100 individuals. If the coefficient k would cause some scaled fitness values in the current population to be negative, the highest possible coefficient that would not do so is used instead.

Other scaling methods include: *sigma truncation*, used to avoid negative values in linear scaling; *power law scaling*, which uses the formula $f' = f^{\alpha}$; and *fitness sharing*, which allows for the development of niches within the population (Goldberg, 1989). While too complex to be discussed here in greater detail, fitness sharing deserves special attention for being dependent not only on the raw fitness values, but also on genotype or phenotype. Its application to grammatical evolution would therefore require an adaptation.

2.6.5 Operators

In addition to the two traditional operators, crossover and mutation, duplication has been proposed for grammatical evolution, but has not gained widespread use. The operators for GE are commonly implemented as follows:

One-point crossover, when applied to a pair of variable-length individuals, selects one point on each of them at random, and exchanges segments starting at these points. The crossover occurs on a codon boundary and the specified probability is used to decide whether the operator is applied to a given pair of individuals. When applied to fixed-length genotype, the crossover occurs at the same point at both individuals (Goldberg, 1989; Koza, 1992, sec. 6.4.2) to maintain the same length. Other types of crossover are described and evaluated by O'Neill and Ryan (2003) with the conclusion that the one-point crossover has superior results.

Bit-level mutation goes through each bit of each individual and inverts it with a specified probability. This type of mutation is the only considered by O'Neill and Ryan (2003). It should be noted that the effective mutation rate is affected by degeneracy (which is determined by the number of bits per codon, length of the chromosomes, structure of the applied grammar and wrapping). It does not make sense to directly compare the values of mutation probability between GE and GP, or even between different GE setups. This version of mutation is one of the most common mutation methods in evolutionary algorithms, it has been described for instance by Goldberg (1989).

Codon-level mutation goes through each codon of each individual and sets it to a random value with a specified probability. This mutation operator has been adopted in GEVA (Section 3.2) without any comments on its benefits or drawbacks compared to the bit-level variant. An obvious advantage is that it has lower computation time: it requires at most two random number generator calls per codon (one to decide whether to mutate, the other to generate a random codon value), while a naïve implementation of the bit-level mutation requires one call per bit. (In Section 5.4 we will show how to overcome this disadvantage.)

Duplication "involves randomly selecting a number of codons to duplicate and the starting position of the first codon in this set. The duplicated codons are placed at the penultimate codon position at the end of the chromosome so as to facilitate their incorporation into the phenotype. We do this because if the the [sic] individual produced a completely mapped program after reading the last codon, and we placed duplicated codons after this point, they will [sic] not be used by this individual, until some other genetic operator allowed them to be switched on." (Citing O'Neill and Ryan, 2003.) This operator was proposed for use in GE by O'Neill and Ryan (2003), where it is applied it to all examples, although only with a 1% probability. A more accurate description as well as an analysis of its effects is missing. Neither of the two major implementations (ligGE and GEVA, see Chapter 3) includes it. It is also omitted from O'Neill's later work on GE (Dempsey et al., 2009). As follows from its definitions, duplication is, in fact, a special mutation operator.

2.6.6 Variations to the algorithm

The basic search algorithm, called simple or generational, presented in Section 2.1 creates a new generation using selection an operators in each iteration. As result of the random nature of its building blocks described above, it may happen that the best individuals from one generation do not make it to the other, while the less fit, or even invalid, individuals may survive. Two variations of the algorithm who remove this possible drawback are commonly used with grammatical evolution. Their descriptions are based on comparison with the simple algorithm defined in the pseudocode for Evolution on page 15.

Steady-state evolution with replacement r $(0 < r \leq 1)$, an alternative to generational evolution, does the following in each iteration:

1. A temporary population Q of $\lfloor r \cdot p \rfloor$ individuals is produced in the same way as a new population of p individuals is produced by the generational algorithm.
2. The temporary population Q and the old population P_i are merged into a new population P_{i+1}.
3. The $\lfloor r \cdot p \rfloor$ worst individuals are deleted from the merged population P_{i+1}.

This is most likely what O'Neill and Ryan (2003) mean by "steady state replacement strategy". Although the authors did not describe any such strategy, the words "Steady State" occur in the parameters of all four of their examples and at least some of the experiments. Only the following two remarks on the application of the steady-state algorithm to GE can be found in the book:

– Page 67 on the examples: "The effective removal of the [invalid] individuals could be attributed [particularly to] the steady state replacement strategy. [. . .] The effective operation of GE is not dependent, however, on using a steady state replacement strategy."

– Page 91 on the experiments with crossover: "[The one-point crossover] produces individuals that are capable of being propagated to successive generations, given the steady state replacement strategy."

Several different algorithms, most of which preserve a certain number of the best individuals from the previous generation, are also referred to as steady-state. This particular version is the most straightforward and is called "GE style" by the authors of libGE (Nicolau and Slattery, 2006), an implementation discussed in Chapter 3.

Generational evolution with elitism does the following in each iteration for elite size e ($1 \leq e < p$, the population size):

1. After the fitness of the old population P_i is evaluated, the e best individuals of P_i are copied to a temporary population E, the elite.
2. A new generation P_{i+1} is created exactly as in the simple, non-elitist algorithm.
3. The elite E is merged into P_{i+1}.
4. The e worst individuals are deleted from the merged population P_{i+1}.

Slight modifications can be made, for instance: the elite may not be allowed to contain more than one copy of the same individual, or only those of the elite E better than the best one of the population P_{i+1} are merged into P_{i+1}.

Chapter 3

Existing Implementations

A GE environment can be built on top of, or as a module of, an existing environment for evolutionary algorithms. This is especially useful for implementing a proof of concept because:

– it spares the developers from reimplementing basic evolutionary algorithms and structures,

– it demonstrates that the principle of grammatical evolution is independent of a specific search algorithm (even algorithms other than genetic can hypothetically be used),

– additionally, it can contribute to a cleaner, more modular design.

However, the GE environment will typically use only a fraction of the underlying infrastructure, and the result will be perceived as clumsy by a user or an application developer. This path has been taken by the authors of libGE, most probably for the reasons cited above (as suggested by Nicolau and Slattery, 2006, sect 2.1).

This chapter will review the major implementations of GE that are publicly available. Unless specified otherwise, names for methods of initialisation, or selection, and variants of operators, or algorithms in the following text are used as defined in Section 2.6.

3.1 libGE

LibGE[1] was developed by the Biocomputing and Developmental Systems group (BDS) at University of Limerick by Miguel Nicolau and Darwin Slattery as a first major publicly available implementation of grammatical evolution. As of this writing it is still being advertised on BDS's website[2], but it has not been updated since September 2006.

As its documentation states (Nicolau and Slattery, 2006), "libGE is a C++ library that implements the Grammatical Evolution mapping process." The relatively small library is meant to be used in conjunction with a separate EA environment.

[1]LibGE source code can be downloaded from `http://bds.ul.ie/libGE/`. The latest published version is 0.26. Documentation for version 0.27alpha1 is available on the website and appears identical to that contained in the 0.26 package.

[2]`http://bds.ul.ie/` and `http://www.grammaticalevolution.org/`

SGA-C[3], GALib[4] and EO[5] are the only EA engines tested with libGE. The package includes three documented example applications with separate implementations for EO and GALib. Two applications are also implemented using SGA-C, but the results are not compared with others due to the limited functionality of SGA-C.

LibGE does not shield the application developer completely from interacting with the search engine. He is expected to choose an engine and write his application by modifying the examples, which comprise actual applications, glue code and extensions to the respective engines.

Chromosome

Chromosome is a string of 8-bit codons, as recommended for simple grammars (see Section 2.4, p. 21) by O'Neill and Ryan (2003). No provisions have been made for changing the codon size.

Algorithm and selection

While libGE itself does not implement any evolutionary algorithm, the examples provide setups for GALib and EO resulting in steady-state evolution with roulette wheel selection. The examples and experiments do not use any other methods.

Initialisation

Both random and "sensible" initialisation is supported.

Operators

The examples include implementations of two simple operators, which are recommended by O'Neill and Ryan (2003): one-point crossover and bit-level mutation. Additionally, a slight variation of crossover, called *effective crossover*, limits the point of crossover to codons actually used when deriving the phenotype. None of the examples implements the duplication operator, which is also recommended by O'Neill and Ryan (2003).

Evaluation

In GALib, fitness evaluation is performed by a set of C-style functions. In EO it is performed by methods of an evaluator class. In either case the fitness function, or the evaluator, must be adapted to the chosen engine, and it is the fitness function, which interfaces with the genotype-phenotype mapper implemented in libGE. All three supported engines maximise fitness. In an example with reverse

[3]SGA-C is a C reimplementation of example code in David Goldberg's *Genetic Algorithms* (1989). Several versions circulate on the internet. The one used in libGE is available at `http://www.illigal.uiuc.edu/pub/src/simpleGA/C/`.

[4]GALib: A C++ Library of Genetic Algorithm Components has a home page at `http://lancet.mit.edu/ga/`.

[5]The EO (Evolving Objects) library can be downloaded from `http://eodev.sourceforge.net/`.

fitness ordering, the raw fitness is simply inverted ($1/x$). In both GALib and EO, the application can, apart from setting fitness, set effective size of the genotype (number of codons actually used in the derivation), guide the "sensible" initialisation, and provide very basic support for printing individuals. This is demonstrated in the examples.

Individuals of a population are evaluated one by one. If a compiled language is used for the phenotype, it means that the compiler is invoked separately for each individual.

Output

Output is limited to what is directly provided by GALib or EO, which means that GE-specific statistics are not available.

Documentation

The documentation concerns chiefly practical questions, which arise from the need to integrate libGE with a GP environment. Results of three example applications in the two supported environments (GALib and EO) are briefly discussed.

Portability

LibGE has been developed in C++ under Linux and can be built using the GNU toolchain (GCC, autoconf, automake). The library itself does not provide any means for phenotype evaluation, but the examples use GCC, TinyCC, Lua, and S-Lang.

3.2 GEVA

GEVA[6] is being developed by Natural Computing Research & Applications Group (NCRA) at University College Dublin. According to its documentation (O'Neill et al., 2008): "It is an open source implementation of Grammatical Evolution [...], which provides a search engine framework in addition to a simple GUI and the genotype-phenotype mapper of GE." While the mapper itself "draws upon the design adopted in Miguel's libGE C++ library," this comprehensive approach distinguishes GEVA from libGE.

Architecture

GEVA uses an abstraction for the evolutionary algorithm, citing from its documentation (O'Neill et al., 2008):

[6]GEVA source code is available at `http://ncra.ucd.ie/Site/GEVA.html`. Current version is 1.1. A technical report (O'Neill et al., 2008), which "serves as an introduction to the GEVA software providing guidelines on its installation and use," is included both in the package and available as a separate download. The report concerns the previous release (1.0), but according to the `changelog` file, version 1.1 is only "a bug fix release which updates the code for the Santa Fe Ant Trail example problem." The package also contains JavaDoc-generated HTML files, which are as of version 1.1 too terse to serve as a documentation.

> In GEVA algorithms are built by combining different modules into a pipeline. A module is a self-contained algorithm building block, by stacking modules an algorithm is created (You could create your entire algorithm in one module, but that is not what GEVA is designed for).

An algorithm, such as simple or steady-state, is not written in a procedural form (see pseudocode for EVOLUTION on page 15), instead it is assembled like a jigsaw by a series of assignments, constructor and accessor method calls. The documentation does not provide any explanation of this design choice. If it was only to make the code modular, it would be at the expense of clarity. My impression is that it could be used to create a very interesting, and maybe effective, graphical interface, but the current GUI, which is merely a form corresponding to the command line options, does not reflect this.

Chromosome

Chromosome is a string of 32-bit codons, which introduces inadequate amount of code degeneracy for any imaginable grammar (see Section 2.4, p. 21). While there is no such explanation in the documentation or the source code, it may be so that the impact of degeneracy on effective mutation rate is minimised by the adoption of codon-level mutation. Further investigation would be needed to understand all implications of this approach, which is significantly different from the one presented and explained by O'Neill and Ryan (2003). No provisions have been made in the source code to change the codon size.

Algorithm and selection

GEVA implements both simple and steady-state evolution, and supports elitism. GEVA's implementation of elitism works as outlined in Section 2.6.6, with the following exceptions: (1) genotypic duplicates are removed from the elite, possibly reducing the initial size e of the elite; (2) steps 3 and 4 are done in reverse order, although the source code comments tell otherwise.

Allowed selection methods are roulette-wheel, tournament and user selection through GUI. All examples use the tournament selection (or, in one case, the user selection). The algorithm is minimising and the only permitted fitness ordering is reverse. As the source code comments put it, "min fitness is the best fitness," and therefore the fitness values go through a mapping that "subtracts the fitness from the fitness sum and divides by the fitness sum" before they are used for the roulette-wheel selection. This makes roulette-wheel selection in GEVA unusable, as these implicitly scaled fitness values all fall in a narrow range, virtually removing any significant fitness pressure.

While other methods of selection could be added easily (although precise documentation is missing), fitness scaling is not supported at all.

Initialisation

Both random and "sensible" initialisation is supported. Random initialisation supports only a fixed chromosome length instead of a range. The two methods are implemented in the classes `Operator.Operations.RandomInitialiser` and

`Operator.RampedFullGrowInitialiser`. In command line options and configuration files, the random initialiser is identified just as `Operator.Initialiser` (a skeleton class that is in fact used by both types of initialisers).

Operators

Crossover uses the standard one-point method, but mutation occurs at the codon level. It is unclear why. Perhaps because it better fits the oversized codons (see above), perhaps to save computation time (see Section 2.6.5). Neither bit-level mutation nor duplication is implemented.

Evaluation

The fitness evaluation is performed by an evaluator class that implements the `FitnessFunction` interface: a `getFitness()` method, which assigns fitness to a single individual, and a `canCache()` method, which can be used to enable caching of the results. (The caching is not documented, but it appears that fitness values are cached in a hash table, where the phenotype strings are used as keys. Least recently used entries are removed when the cache reaches a certain size.)

Two abstract classes that implement the interface are available: one evaluates phenotype with JScheme (a dialect of Scheme), the other with Groovy (an OOP language for the Java Platform) using the Bean Scripting Framework. Both of them require the application developer to supply a single method, `createCode()`, which accepts a phenotype string, and returns a program in the given language, which in turn returns the final fitness value. As a result the whole fitness evaluation has to be implemented in the scripting languages used for phenotype evaluation. (The evaluator can be also implemented from scratch and use the JScheme, Groovy, or another interpreter as needed.)

Use

GEVA is designed to perform a single run of an evolutionary algorithm set up via command line interface (CLI) or graphical user interface (GUI). The documentation lists command line options shown in Listing 3.1. The same options can alternatively be set up in a configuration file or using GUI. An undocumented option worth mentioning is `-rng_seed`, which can be used to set a seed for the random number generator. Otherwise it is seeded with the current system time, making it impossible to reproduce the results.

These options, although most of them are not described in more details in the documentation, are easy to understand and are sufficient to configure the algorithms implemented in GEVA. A technical detail worth mentioning is that the `-generation` option, which according to the documentation, should set the (maximum) "number of generations" (g, as used in this thesis, by Koza, 1992; Banzhaf et al., 1998; and also by O'Neill and Ryan, 2003), actually sets the maximum generation number ($maxgen = g - 1$, as used by Goldberg, 1989). Generations are counted starting from zero as usual.

The option `-stopWhenSolved` does not work as expected in most cases, as it depends on the fitness value reaching exactly zero (see the discussion in footnote 10 on page 23). For instance in the first three sets of runs of simple symbolic

```
-mutation_probability              -fitness_function
-initialiser                       -max_wraps
-generations                       -selection_operation
-replacement_type                  -crossover_probability
-generation_gap                    -mutation_operation
-crossover_operation               -max_depth
-evaluate_elites                   -elite_size
-userpick_size                     -word
-grammar_file                      -tournament_size
-grow_probability                  -fixed_point_crossover
-population_size                   -stopWhenSolved
-output                            -rng_seed (not documented)
```

Listing 3.1: GEVA options, as listed in the documentation and help.

regression presented in Section 8.2 the fitness value of zero was reached only in 2 cases out of the total 908, in which the solution was found.

Notably absent from the options are output settings. One can only choose whether to save the output to a file or not.

Nonexistent options or options not used by the chosen algorithm elements are silently ignored. Missing options are either supplied from the "hello world" example, or cause an exception, usually uncaught. It does not pose problems when using the GUI, which constraints the choice of parameters, but it is a highly unreliable behaviour for a command line tool.

Output

GEVA is designed to perform one run of its evolutionary algorithm and progressively output basic statistical data about each generation. The generation statistics are best fitness, average fitness, fitness variance, average chromosome length, average number of used codons, average derivation tree depth, elapsed time[7], and number of invalid individuals. These statistics can be printed to standard output, displayed as graph (when GUI is used) or logged to a file as space-separated values (using the `-output` option). When finished it prints the best individual's phenotype string, fitness value, and total elapsed time to the standard output. The graph in the GUI version is plotted on the fly to provide visual feedback during the run, it can also be scaled and saved as a bitmap image.

No other statistics or output options are provided. It is impossible to retrieve information about any individuals other than the best of run. GEVA also does not provide any means for aggregating statistics from multiple runs.

Documentation

The documentation introduces GEVA through a series of task-oriented tutorials for application developers and users. It does not, however, document the

[7]GEVA only measures the elapsed time (based on the "wall clock time" as supplied by `System.currentTimeMillis()`), as opposed to its consumed CPU time. The reported times therefore cannot be used for benchmarking.

general design or the inner workings of GEVA. It does not state clearly what algorithms and methods are used, except for the genotype-phenotype mapping using a grammar explained in the introduction.

Portability

GEVA is written in Java and does not directly depend on any other software, except for several open-source Java libraries, which are included with GEVA.

3.3 Other implementations

Michael O'Neill maintains a list of grammatical evolution software[8]. GEVA and libGE are followed by jGE[9], PyNeurGen[10], and DRP[11]. According to their web sites, DRP implements a new generative programming technique, a cross between grammatical evolution and genetic programming, and PyNeurGen combines grammatical evolution with neural networks. While interesting, they are not general tools for GE. The jGE project on the other hand implements grammatical evolution in a minimalistic and clean way. The source code is well documented using JavaDoc. Unfortunately it is the only documentation provided. There has not been any update since the release of version 1.0, and the web site was last updated in March 2008.

As of this writing there are no other publicly released implementations of GE I know of.

3.4 Conclusion

Most of the research of grammatical evolution to date has been connected with the UCD NCRA, Natural Computing Research & Applications Group at University College Dublin (authors of libGE, Michael O'Neill, Anthony Brabazon), and the BDS, Biocomputing and Developmental Systems group at University of Limerick (authors of GEVA, Conor Ryan, formerly Michael O'Neill).

While there are several software projects that implement grammatical evolution, or are based on it, only GEVA both aims to provide a comprehensive EA environment and is actively developed. I will therefore focus on comparison with GEVA in the following chapters. When appropriate I will also refer to libGE, from which GEVA derives its implementation of genotype-phenotype mapping.

[8]Michael O'Neill's list of GE software: `http://www.grammatical-evolution.org/software.html`

[9]jGE web site: `http://www.bangor.ac.uk/~eep201/jge/`

[10]PyNeurGen web site: `http://pyneurgen.sourceforge.net/`

[11]DRP web site: `http://drp.rubyforge.org/`

Chapter 4

Goals

Goals of the accompanying software project are based on both theoretical background of GE and evolutionary algorithms, as presented in Chapter 2, and the analysis of strong and weak points of available implementations in Chapter 3. The software is named AGE (Algorithms for Grammatical Evolution) to reflect that it does not implement only grammatical evolution as a genotype-phenotype mapping, but also other related algorithms. I have set the following goals, which will be reviewed in this chapter:

- a clean, comprehensive implementation of standard algorithms,
- modularity,
- adequate documentation,
- versatile output,
- reproducible results,
- acceptable performance.

Graphical user interface is not among the goals, instead focus has been put on the quality of the command line interface and textual output. A simple GUI such as implemented in GEVA, would be a valuable addition that would let AGE serve better as a demonstration or a tool for exploration of grammatical evolution for a casual user. Having said that, it is of far greater importance that AGE can serve as a research tool.

4.1 Implementation of standard algorithms

As has already been proved by libGE, it is possible to implement grammatical evolution independently of evolutionary algorithms. AGE tries to preserve that separation where it contributes to cleaner design and permits better future extensibility. It implements, however, a complete environment for evolutionary algorithms in a similar way as GEVA does. This has foremost the advantage that the software can be used as a simple, consistent tool. Additionally, it allows for more straightforward implementation of GE-specific features such as output of statistics related to the genotype-phenotype mapping.

The general algorithms and techniques include all those commonly used with grammatical evolution (see Section 2.6). I have made the selection of algorithm elements both to match the extent of other GE implementations, and based on descriptions by O'Neill and Ryan (2003). The implementation itself is based on a broader understanding of evolutionary algorithms, and draws from respected sources (Goldberg, 1989; Koza, 1992; Banzhaf et al., 1998; Poli et al., 2008) when possible. I have not attempted to replicate other implementations. This allowed me to make better sense of all the techniques I implemented, and I hope it will help the users of AGE likewise. Any particular algorithm used in AGE can be identified, traced to its description in literature, and easily compared to other (documented) implementations.

The genotype-phenotype mapping is implemented according to O'Neill and Ryan (2003).

4.2 Modularity

All algorithm elements are implemented in such a way that they can be easily changed or replaced. Like the authors of GEVA I have made extensive use of abstract classes (C++ equivalent of interfaces in Java). Unlike them I do not use any abstraction of the evolutionary algorithm itself. I believe that this makes for more readable and clearer code while not compromising modularity. When there is a real need for a pipeline-based approach, it can be incorporated easily, thanks to the modular design of all classes.

4.3 Documentation

User documentation provides an overview of all features, including the API (application programming interface), which can be used to implement custom modules and applications. It also features a tutorial for that purpose. All algorithms accessible from the command line tool and using the API are clearly defined.

4.4 Output and results

As is usual for a command line tool AGE outputs both the final result and intermediate results at specified intervals to provide the user with immediate feedback. More detailed data, optionally including information about all individuals, can be saved in files structured using XML. This is a much more robust and forward compatible way of storing data than simple delimiter-separated values. XML output is easy to analyse with external tools, and can also visualised using a XSL style sheet.

The results must be reproducible. They are entirely based on the supplied parameters, including a seed value for a random number generator (RNG). AGE uses a portable reentrant RNG implementation, which produces deterministic sequences even in a threaded environment. The parameters are always saved along with results.

An evolutionary algorithm with given settings can be conveniently executed several times with a defined sequence of RNG seeds. In that case, aggregate statistics from all runs are generated in addition to the per-run output.

4.5 Performance

Although optimal performance is not a primary goal, AGE aims to be a practical tool, not a demonstration, and therefore performance issues cannot be ignored. To ensure that the implementation is competent, performance in benchmark applications will be examined (in Chapter 8). Additionally, AGE supports parallel execution of several runs of an evolutionary algorithm, making it possible to employ multiple CPUs or CPU cores simultaneously.

Chapter 5

Design Decisions

This chapter describes the overall design of AGE in Section 5.1, and highlights a few of its parts interesting for their design:

- the ramped initialisation method, a generalisation of the “sensible” initialisation method, in Section 5.2,
- the evaluation of individuals, performed in C and Lua, in Section 5.3,
- the implementation of mutation operators, including a fast implementation of bit-level mutation, in Section 5.4.

In Section 5.5 we discuss our approach to portability.

5.1 Overall design

AGE has a simple component-based design shown on Figure 5.1. Its central part is an engine for evolutionary algorithms and grammatical evolution, the *EA/GE engine.* This engine implements an evolutionary algorithm suitable for genetic programming and the related data structures, algorithms and data structures for grammatical evolution, and an environment for execution of multiple runs of the algorithm.

The *algorithm elements* (initialisers, operators, selectors, and fitness scalings) are implemented as separate components that communicate with the EA/GE engine through a documented API (application programming interface). Components corresponding to all algorithm elements described in Section 2.6 are already implemented. *Applications*, represented by a fitness evaluator, can be implemented in the same way, as components using the API. A tutorial in Section 6.6, and the applications used for experiments (Chapter 8) demonstrate this.

The EA/GE engine does not produce any human-readable output by itself, it passes its results to the *output engine*, which filters, aggregates, and formats the results. Some of the output is sent to the console for immediate user feedback, while the more detailed data are stored in files for later processing.

At the top of these components is a simple *command line tool.* It sets up the EA/GE engine, the algorithm elements, the application, and the output engine according to user options. It ensures that the supplied options are correct and sufficient to set up a consistent evolutionary algorithm. Both the algorithm elements and the applications can advertise their user-configurable parameters

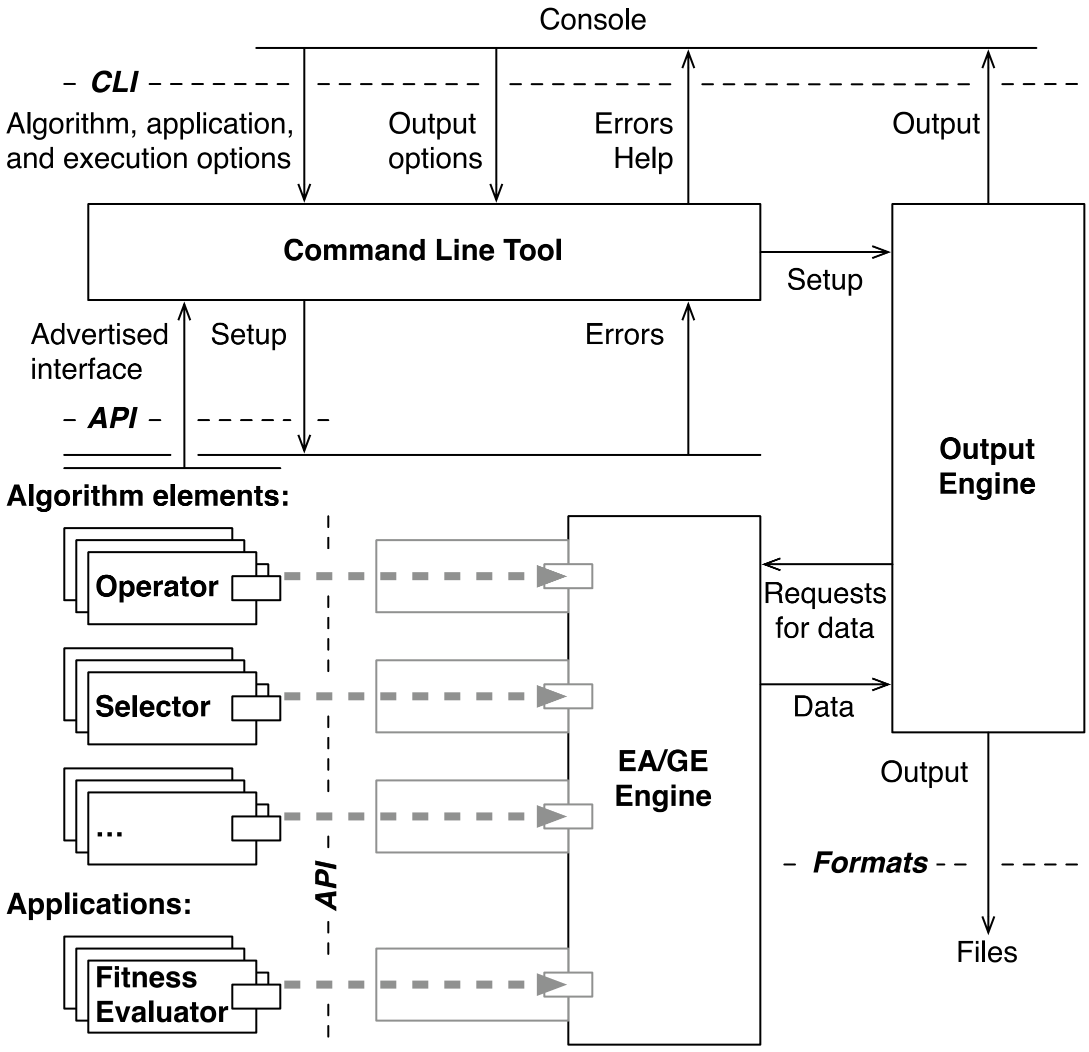


Figure 5.1: Overall design of AGE, interfaces marked with a dashed line: CLI (command line interfaces), API (application programming interface), and file formats.

to the command line tool and use an infrastructure for checking the parameters and reporting errors. This is a simple way to provide a consistent user interface across all, existing and future, components. The API that makes this possible is separate from the core engine API. It is also general enough to be used to construct a different, currently unsupported, interface (configuration files, GUI forms) without any changes to the components.

As can be seen on Figure 5.1, this design leads to four interfaces:

– the command line interface,

– the output file formats,

– the EA/GE engine API,

– the (command line) tool API.

These interfaces, as well as the implemented algorithm elements, are documented in the user documentation in Chapter 6.

5.2 Ramped initialisation

AGE implements a slight generalisation of the "sensible" initialisation method described by O'Neill and Ryan (2003, sec. 8.8), which is based Koza's ramped half-and-half initialisation (Koza, 1992, sec. 6.2, app. C).

O'Neill and Ryan (2003) define *recursive* production rules as production rules that can be used to derive a tree of arbitrary height, and state: "In Koza's GP, functions are responsible for the growth of a tree. As GE uses grammars, this role is taken over by the *recursive* production rules. [...] If we are using the full method, if possible, we only choose recursive rules." However, if we accepted that the recursive productions are the GE equivalent of GP functions, then (1) the same argument could be applied to the "grow" (as opposed to "full") method as well; (2) it would follow that the non-recursive productions correspond to GP terminals, and therefore the derivation subtree that starts with these productions should count only for one (the last) level of tree height.

Taking these thoughts one step further, it becomes clear that production rules, recursive or not, do not correspond to GP functions at all. In fact the terminals that represent functions or operators are the equivalent of GP functions, while nonterminals and production rules depend entirely on the structure of the grammar. Along these lines, it would be more correct, to first convert the grammar to a form in which all production rules rewrite a nonterminal either to terminals or to a string containing a terminal corresponding to a GP function. Then an increase in a GE derivation tree height would correspond exactly to an increase in a GP tree height.

This is not to say that the derivation tree height cannot be successfully used as an approximation of the *relative* complexity of the derived string for *most* grammars. I do not, however, see any added benefit in the special treatment of recursive productions when using the full method.

On these grounds, the ramped initialisation method that I have implemented (**`RampedInitialiser`**, see Section 6.3.1) supports this special treatment of recursive productions only optionally. It can be enabled by the parameter `oneill`. Additionally, I have implemented optional generation of unique trees, briefly suggested by O'Neill and Ryan (2003), which is present neither in libGE nor in GEVA. I have modelled it after the same feature used in GP by Koza (1992). For its purposes two individuals are considered equal if they map to the same sequence of production rules. AGE also supports other, rather technical, parameters to match the functionality provided by GEVA or libGE (see Section 6.3.1).

The observations made in this short discussion would deserve further development, and could be applied to other parts of grammatical evolution as well.

5.3 Evaluation in C and Lua

The fitness evaluation in grammatical evolution is a three-stage process. The genotype is mapped to the phenotype using a grammar, the resulting phenotype string is then interpreted as a program in some language, and finally a fitness value is assigned to the output. Unlike in traditional GP, virtually any programming language can be used. AGE does not constrain this choice. Without any additional configuration AGE supports evaluation in two languages that cover a wide range of applications. Interpreted code will have better performance in most cases: as candidate solutions tend to be short and are executed only a fixed number of times (for each fitness case), the cost of interpreting a program is smaller than the cost of compilation and the overhead of launching a separate process. On the other hand, if the candidate solutions perform intensive computations, the faster execution of compiled code will outweigh these costs.

Compilation is supported using a C compiler. It can work directly with GCC (GNU Compiler Collection) and PCC (Portable C Compiler), support for other compilers can be added easily. In both cases the whole population is compiled as a single unit, common code can be compiled once for all individuals, and the population is then executed once as a single process even if multiple fitness cases are evaluated. The source code, the program input and output is sent using a pipe, instead of using temporary files.[1] Additionally, with GCC it is possible to disable the use of temporary files for intermediate stages of compilation. These measures reduce the overhead of compilation and execution to a minimum.

Interpreted code is supported using Lua, a lightweight scripting language. Lua is fast, portable, and easy to learn thanks to its simple syntax.[2] Its small code base and its permissive open-source licence (see Section 6.7) make it possible to distribute Lua together with AGE.

5.4 Mutation operators

The role of mutation in evolutionary algorithms is usually regarded as secondary, and the mutation rates are therefore set accordingly low. Koza (1992, p. 27) goes as far as using a mutation rate of 0 in all his examples, other authors suggest mutation probabilities ranging from 0.1 % (Goldberg, 1989) to 10 % (Banzhaf et al., 1998), depending on the problem and the mutation method adopted.

In Section 2.6.5 we have said that the bit-level mutation goes through each bit of each individual and inverts it with a specified probability. The most straightforward implementation does exactly that (Goldberg, 1989, fig. 3.7), which involves calling a random number generator (RNG) once for each bit of genotype, regardless how small the mutation probability might be. Given the relative complexity of random number generation, this can significantly contribute to the computational time of an evolutionary algorithm.

To remedy this inconvenience, I have implemented a fast version of the bit-level mutation. As Goldberg (1989) remarks, "it would be possible to avoid much random number generation if we decided when the next mutation should occur." Indeed, as the decision whether to mutate a single bit follows a Bernoulli

[1]The current implementation uses a POSIX socket pair, which works reliably on several different UNIX systems. Bidirectional pipes or any other mechanism of interprocess communication that can be attached to standard input and output streams can be used instead.

[2]Citing the home page of Lua (`http://www.lua.org/about.html`): "Several benchmarks show Lua as the fastest language in the realm of interpreted scripting languages." This of course depends on the choice of benchmark tests and other factors. An independent continually updated comparison of benchmark results of various computer languages is available at `http://shootout.alioth.debian.org/` (*The Computer Language Benchmarks Game*), where Lua is ranked higher than the popular scripting languages Python, PHP, Perl, and Ruby. A website dedicated to Lua offers more information on how it compares to other languages: `http://lua-users.org/wiki/LuaComparison`.

distribution with parameter p_m (the probability of mutation), the number of unaltered bits until the next mutation follows a geometric distribution with the same parameter. A random variate from geometric distribution can be generated in constant time using one RNG call. The implementation details are described in Section 7.2.

As this version of bit-level mutation is statistically equivalent to the naïve implementation, I use it as the default bit mutation operator. For the sake of completeness the naïve implementation is also available.

The codon-level mutation is implemented in exactly the same way as in GEVA, so that comparison of the two mutation operators can be made. The duplication operator is implemented according to its description by O'Neill and Ryan (2003) (see Section 2.6.5), as far as it was possible to understand it.

I am unaware of any reason why the bit-level mutation (used by O'Neill and Ryan, 2003), or the codon-level mutation (used in GEVA: O'Neill et al., 2008) should be particularly suitable for grammatical evolution, except for their easy implementation. A common trait of these two operators is that due to genetic code degeneracy, there is a high probability that their application does not affect the phenotype at all, yet once it does, it has a potential of changing the phenotype very significantly. Other kinds of genetic programming tend to use operators with more consistent behaviour and effects restricted to a small part of the phenotype, usually a single subtree (Banzhaf et al., 1998; Koza, 1992).

It seems that the methods of mutation are a neglected area of grammatical evolution. Both the duplication operator (see Section 2.6.5) and the codon-level mutation operator can be seen as attempts to find a mutation method that would better suit grammatical evolution, although poorly rationalised ones.

I will compare bit-level mutation and codon-level mutation in Chapter 8.

5.5 Portability

One of the most important goals of this project is the reproducibility of results. As a consequence, the implementation must be portable across platforms compliant with a defined set of standards. AGE is implemented in C++ and C, languages that provide a reasonable trade-off between portability and performance, which is another, albeit less critical, goal we have set.

There are only two conditions *necessary* for compiling and running AGE:

- a C and C++ language compiler and interfaces as defined in the ISO/IEC 14882:1998 (informally "C++ 89") and ISO/IEC 9899:1990 (informally "C 90", virtually identical to ANSI X3.159–1989, "ANSI C") standards,
- single (32-bit) precision and double (64-bit) precision binary floating-point data types, accessible as `float` and `double`, conforming to the IEEE 754–1985 standard.

AGE is distributed together with the scripting language Lua (see Section 5.3), which has a similar policy regarding standards, and which can be compiled with any compiler and library that supports "C 90".

Additionally, AGE has several *optional* features, which can be enabled in environments at least partially conforming to the POSIX (IEEE 1003.1) standard.[3] Whether these features are used or not, does not affect the reproducibility of results in any way. AGE has been successfully tested on recent versions of Mac OS X, NetBSD, and Linux.

[3]POSIX (Portable Operating System Interface for Unix) defines common features of modern Unix systems such as file system operations, concurrency or interprocess communication. Compliant systems include Mac OS X (fully compliant), Linux (partially compliant), and Microsoft Windows (partially compliant using the free Cygwin environment or Microsoft's own Windows Services for UNIX).

Chapter 6

User Documentation

The user documentation describes the build and installation procedure in Section 6.1, and covers the interfaces of AGE:

- the command line interface in Section 6.2; the implemented components, which are available from the command line, in Section 6.3;
- the output file formats in Section 6.4;
- the application programming interfaces in Section 6.5; a tutorial for application developers in Section 6.6.

Section 6.7 contains copyright and licence information. The user documentation, including the documentation of the API, is part of the thesis because there are frequent references between the documentation and the rest of the text.

6.1 Building and installation

AGE is distributed as a portable source code package, and can be built and installed either automatically using the standard GNU or BSD `make` utility or manually. Both procedures involve building the included Lua scripting language library. The final products are the following files:

- `libAGE.a`, `liblua_AGE.a`, static libraries, usually installed in *prefix*`/lib`,
- header files for the AGE library, usually installed in *prefix*`/include/AGE`,
- `AGE`, a command line tool, usually installed in *prefix*`/bin`,
- data files, usually installed in *prefix*`/share/AGE`,

where *prefix* is the installation directory prefix, such as `/usr/local`.

Automatic build procedure

Change the working directory to the top level of the distribution. To build AGE without installing, invoke `make` with the default target `all`:

```
% make all
```

—or alternatively without arguments—

```
% make
```

To build and install in the current directory, creating the usual subdirectories `bin`, `include`, `lib`, and `share`, invoke `make` with the target `local`:

```
% make local
```

To build and install into a specified installation directory prefix, invoke `make` with the target `install` and define the `INSTALL_PREFIX` variable:

```
% make INSTALL_PREFIX=prefix install
```

You may choose to install into your home directory or to `/usr/local` using `sudo` to acquire superuser rights:

```
% make INSTALL_PREFIX=~ install
% sudo make INSTALL_PREFIX=/usr/local install
```

If the build process fails because of undefined or conflicting symbols, you can try to rebuild AGE with limited features to conform strictly to "C++ 89" and "C 90" (see Section 5.5) by specifying the target `ansi` (for Lua) and the preprocessor macro `AGE_ANSI` (for AGE itself, target `all`):

```
% make clean
% make EXTRAFLAGS=-DAGE_ANSI ansi all
```

If your system does not contain some of the tools used in the makefile, consult the comments in `Makefile` and `src/Makefile`. For more build options, invoke:

```
% make help
```

Manual build procedure

When an appropriate `make` utility or other tools used in the makefile are not available on your system, you can build AGE manually.

Library: To build the library (AGE and Lua combined), compile and link the following source files as a static library:

- Lua C source files located in `src`: `lapi.c`, `lcode.c`, `ldebug.c`, `ldo.c`, `ldump.c`, `lfunc.c`, `lgc.c`, `llex.c`, `lmem.c`, `lobject.c`, `lopcodes.c`, `lparser.c`, `lstate.c`, `lstring.c`, `ltable.c`, `ltm.c`, `lundump.c`, `lvm.c`, `lzio.c`, `lauxlib.c`, `lbaselib.c`, `ldblib.c`, `liolib.c`, `lmathlib.c`, `loslib.c`, `ltablib.c`, `lstrlib.c`, `loadlib.c`, `linit.c`, located in `src/lua-5.1.4/src`,
- AGE C source files located in `src`: `BidiPipe.c`, `PathUtils.c`, `Random.c`, `getopt.c`,
- AGE C++ source files located in `src`: identified by the `.cpp` extension, except `main.cpp` and those with the prefix `App-`.

Set your header search paths to `src`, `src/lua-5.1.4/src`, and `src/lua-5.1.4/etc`.

Command line tool: To build the command line tool, compile and link the C++ source files `main.cpp` and those with the prefix `App-`, located in `src`, and the static library from the previous step as an executable file. Search for headers in the same directories.

Depending on the environment you use, you may want to define some of the following macros:

- `NDEBUG` is the standard macro to disable C assertions.
- `LUA_USE_POSIX` enables the use of POSIX interfaces for Lua, otherwise interfaces are restricted to those defined by "C 90".
- `AGE_ANSI` restricts the interfaces used by AGE to those defined by "C++ 89" and "C 90" (see Section 5.5), disables the use of POSIX interfaces among others. If the POSIX thread library is not available on your system, you will also need to remove it (`-lpthread`) from the linker options (the `LIBS` variable in `src/Makefile`).
- `AGE_CODON_SIZE=`*n* sets the size of codons in bits. The default value is 8. Accepted values are between 1 and 31. Codons are aligned on the boundary of the smallest unsigned integer type sufficient to hold *n* bits.
- Other macros, described in the header files `src/Configuration.h` and `src/lua-5.1.4/luaconf.h`.

```
ArgUtils.h                         HashSet.h
BNF.h                              Individual.h
BNFGrammar.h                       IndividualCollection.h
Base.h                             IndividualDescriptor.h
CCodeLauncher.h                    Initialiser.h
CGrammaticalCodeGenerator.h        LivePopulation.h
CLivePopulation.h                  LuaCodeLauncher.h
Code.h                             LuaGrammaticalCodeGenerator.h
CodeFitnessEvaluator.h             LuaLivePopulation.h
CodeGenerator.h                    Operator.h
CodeLauncher.h                     Operators.h
CodonVector.h                      Populable.h
Configuration.h                    PathUtils.h
Elements.h                         PtrSet.h
Error.h                            RampedInitialiser.h
FitnessEvaluator.h                 Random.h
FitnessScaling.h                   RandomInitialiser.h
FitnessScalings.h                  Selector.h
FitnessUtils.h                     Selectors.h
Grammar.h                          Tool.h
GrammarChoice.h                    pstdint.h
GrammaticalCodeFitnessEvaluator.h  yasper.h
GrammaticalCodeGenerator.h
```

Listing 6.1: AGE library header files, all located in `src`.

Other files: Whether you build a custom executable or the default command line tool, you will also need to put the data files located in `resources/AGE` to a proper place. When POSIX interfaces are available, AGE expects them in `../share/AGE` relatively to the executable as usual on Unix systems, otherwise it expects them in the current directory.

To use the static library for a custom executable, you will need to copy the C++ header files from Listing 6.1 located in `src` to your header search path.

6.2 Command line interface

The AGE demonstration command line tool, or a custom executable based on the **`Tool`** (see Section 6.5.8) class from the AGE library, is controlled by command line options, and outputs the results to the console. This section describes the options and the format of the console output. The file output is described separately in Section 6.4.

The options enclosed in brackets are not mandatory. The order of the options does not matter. When no options are supplied, a short summary of options is printed, including the possible arguments for the options **`-X`**, **`-M`**, **`-I`**, **`-A`**, **`-s`**, and **`-S`**.

Unless indicated otherwise, the following convention is used for option arguments:

– n is a natural number,
– p, or r, is a probability, or a rate, from $[0, 1]$,
– f is a fitness value.

Custom executables may have additional options or supply different default values, possibly making different options mandatory.

Algorithm options

The arguments accepted by the options **`-X`**, **`-M`**, **`-I`**, **`-A`**, **`-s`**, and **`-S`** depend on the available components (algorithm elements and applications). Invoking the tool with no options lists the available arguments and provides a brief synopsis of each. The components already implemented in AGE, and thus available in the demonstration command line tool, are fully documented in Section 6.3.

For information about implementing your own components or customising the command line tool, see Section 6.5.

`-p` n sets the number of individuals in the population to n.

`-g` n sets the maximum number of generations to n. The evolution starts with generation 0 and proceeds up to generation $n - 1$.

`-x` p sets the probability of the crossover operator specified by the **`-X`** option to p.

[**`-m`** $p_1, \ldots, p_n$] sets the probabilities of the mutation operators, in the order they are listed in the **`-M`** option to $p_1, \ldots, p_n$, a comma-separated list. The option is mandatory if mutation is enabled.

[**`-r`** r] enables the steady-state algorithm with replacement rate r (see Section 2.6.6).

[**-f** *f*] stops the evolutionary algorithm when the best individual of the population reaches the fitness level *f* or better.

[**-X** *Crossover*] specifies the crossover operator with the argument *Crossover*. Default: one-point crossover (`-X one-point`).

[**-M** $M_1, \ldots, M_n$] specifies the mutation operators with the comma-separated arguments $M_1, \ldots, M_n$. Default: fast bit-level mutation (`-M bit`).

-I *Initialiser* specifies the initialiser with the argument *Initialiser*.

-A *Application* specifies the application, represented by a fitness evaluator, with the argument *Application*.

[**-s** $s_1, \ldots, s_n$] specifies the fitness scalings with the comma separated arguments $s_1, \ldots, s_n$. The scalings are applied in that order. Default: no scaling.

[**-S** *Selector*] specifies the selection scheme with the argument *Selector*. Default: roulette-wheel (`-S roulette`).

[**-e** *e*] [**-d**] [**-b**] enables elitism with an elite of *e* individuals (see Section 2.6.6). Unless the **-d** option is supplied, multiple instances of the same individual are removed from the elite, *e* is then the initial size and may be reduced if duplicates are present. If the **-b** option is supplied, only those individuals from the elite that are better than the current best individual of the generation are merged back into the population.

Execution options

[**-n** *n*] executes a sequence of *n* runs of the evolutionary algorithm set up by the above options. Default: execute one run and seed the random number generator directly with the value specified by the **-R** option (`-n 0`).

[**-R** *n*] seeds the random number generator with the number *n*. If a sequence of runs is enabled by the **-n** option, the number is used to seed a generator that generates seeds for the runs. (The generated seeds can be found in the XML output, see Section 6.4.1.)

[**-t** *n*] creates a thread pool of *n* execution threads. Each thread executes one run at a time. For optimal results, do not set the number of threads greater than the number of runs or greater than the number of available CPUs or CPU cores. The number of threads does not affect the results. Default: no thread pool, only the main thread (`-t 1`). Threads are available only if AGE is compiled with POSIX threads enabled.

Console and file output options

The following options affect the console output and output to the text files (described in Section 6.4.2). The XML output (described in Section 6.4.1) is not affected.

[`-h` *f*] records a *hit* when the best individual of a population reaches the fitness level *f* or better. This option is similar to `-f`, except that it only affects the output, and does not stop the evolutionary algorithm. If the option is set, the number of hits per generation accumulated from all runs is printed to the console and to the file **`GenerationHits.txt`** in the output directory, if specified by the **`-o`** option. The format of the file is described in Section 6.4.2.

[`-v`] enables moderately verbose output:
- run results in the console output will contain the total numbers of operations performed by each operator;
- generation results (activated by `-G`) will contain additional fields;
- overall statistics file **`GenerationStats.txt`** (activated by `-o`) will also contain these additional fields;
- overall statistics in the console output (activated by `-O`) will also contain these additional fields.

The latter three share the format of the **`GenerationStats.txt`** file described in Section 6.4.2, which also lists the additional, “verbose”, fields.

Console output options

[`-G` *n*] prints statistics from each *n*th generation of every run. When used in conjunction with threads (the `-t` option), it intermixes output from multiple runs.

[`-O`] prints statistics of all runs aggregated by generation. Regardless of this option the overall statistics are saved in the file **`GenerationStats.txt`** in output directory, if specified by the **`-o`** option. Also see `-v`.

File output options

[`-o` *OutputDir*] saves XML output and statistics into directory *OutputDir*. If AGE is compiled with POSIX interfaces enabled, nonexistent directories are created, and directories already containing output are not overwritten.

[`-i`] includes details about each individual in the XML output.

The file formats are described in Section 6.4.

Console output

The console output serves as immediate feedback and overview of the results. It does not contain any information that could not be retrieved from the file output, which is more suitable for further processing.

By default, AGE prints the best individual of each run of the evolutionary algorithm and only the most important related information:

- Whether the fitness stop limit has been reached (the `-f` option).

- Run: runs are numbered from 0 to $n-1$ in a sequence of n runs. When threaded execution is enabled, runs with lower numbers may finish earlier than runs with higher numbers.
- Best fitness: fitness value of the best individual found during the run.
- Generation: first generation (numbering from 0) containing the best-of-run individual. If the fitness stop limit has been reached, it is also the last generation executed in that run.
- Individual: the best individual. Its representation is application-dependent, by default, the phenotype string is printed.
- Optionally, the total numbers of operations performed by each operator, activated by the `-v`, "verbose", option.

Example of such an output of four runs can be seen in Listing 6.2. In runs 2 and 3, there were no improvements in the best fitness value after generation 54 and 8 respectively, and the two runs continued up to to maximum number of generations, 101, without reaching the desired fitness level 10^{-5}. In runs 0 and 1, the desired fitness level has been reached, and the runs were stopped at generation 15 and 25 respectively.

Optionally, the console output can contain two additional sections, which are printed out after all runs finish:

- cumulative fitness hits, activated by the `-h` option, which replicate the contents of the `GenerationHits.txt` file,
- averaged overall statistics, activated by the `-O` option, which replicate the contents of the `GenerationStats.txt` file.

If activated, these section are included in the console output, even if the file output is not enabled (the `-o` option). Listing 6.3 shows such an output. Note that the option `-f` used for Listing 6.2 has been replaced by the `-h` and `-O` options. Format of these sections is described in details in Section 6.4.2

Finally, when an error occurs or the user supplies incorrect options, AGE terminates with a nonzero exit value, and prints an error message to the standard error stream. Any unfinished XML output is terminated properly and will include an appropriate `stopped` element as defined in Section 6.4.1.

```
% AGE -n4 -p100 -g101 -e1 -x0.9 -m0.02 -I'random(10-99)' -Areg -sadj -R55
-f1e-5

 S:   1 if fitness stop limit (1e-05) reached, 0 otherwise
 Gen: generation when BestFitness of the run first recorded

S Run   BestFitness  Gen   Individual (best of every run)
1     0  2.75215e-31    15 ( ( ( x * ( x * x ) ) + x ) * ( 1.0 + x ) )
1     1  4.01749e-31    25 ( ( x * ( ( 1.0 + ( ( 1.0 + x ) * x ) ) *
 x ) ) + x )
0     2     0.507607    54 ( ( x + 1.0 ) * ( x + ( x * x ) ) )
0     3     0.507607     8 ( ( ( ( ( 1.0 + x ) * x ) + 1.0 ) + x ) * x )
```

Listing 6.2: Console output example: minimal output.

```
% AGE -n4 -p100 -g101 -e1 -x0.9 -m0.02 -I'random(10-99)' -Areg -sadj -R55
-h1e-5 -O

 S:   1 if fitness stop limit (unset) reached, 0 otherwise
 Gen: generation when BestFitness of the run first recorded

S Run   BestFitness  Gen   Individual (best of every run)
0     0  2.14163e-31    95 ( ( ( 1.0 * ( x * x ) ) + x ) * ( 1.0 + ( ( x
 * x ) * 1.0 ) ) )
0     1  3.76327e-31    51 ( ( x * ( ( 1.0 + ( ( ( x * ( ( x * x ) +
 x ) ) + 1.0 ) + x ) ) * 1.0 ) ) - x )
0     2     0.507607    54 ( ( x + 1.0 ) * ( x + ( x * x ) ) )
0     3     0.507607     8 ( ( ( ( ( 1.0 + x ) * x ) + 1.0 ) + x ) * x )

CUMULATIVE FITNESS HITS:
Gen   Runs with fitness better than or equal to 1e-05
    0     0
    1     0

    ⋮

   14     0
   15     1

    ⋮

   24     1
   25     2

    ⋮

  100     2

AVERAGED OVERALL STATISTICS:
Gen   Invalids     BestFitness  AvgFitness
    0      13.5000      6.35023      43.9313
    1      4.25000      3.06787      21.7976

    ⋮

  100      1.25000     0.253803      46.8393
```

Listing 6.3: Console output example: output with fitness hits (shortened).

6.3 Implemented components

This section mirrors the structure of Section 2.6, as the AGE library implements all algorithms described in that section. Additionally, we will describe the implemented applications, which are part of the demonstration command line tool.

Each component is listed under the name of the class that implements it, which is followed by a description and three short sections:

Argument: The argument form to be used for the corresponding command line option in order to specify an algorithm element. Arguments that do not have any options are simple labels, such as **`roulette`** for roulette-wheel selection. A roulette-wheel selection is therefore specified by `-S roulette`. Arguments that have parameters are in one of the following forms:

$$\textit{label}(m_1, \ldots, m_M)$$
$$\textit{label}(m_1, \ldots, m_M[, o_1, \ldots, o_N])$$
$$\textit{label}([o_1, \ldots, o_N])$$

where *label* identifies the algorithm element, m_1 to m_M are mandatory parameters, and o_1 to o_N are optional parameters, which have default values. Parentheses and commas are part of the syntax. The parameters are positional, which means that you can supply the first $n \leq N$ of the optional parameters, but not for instance only the first one and the third one. If there are no mandatory parameters and you do not supply any optional ones, the enclosing parenthesis can be omitted. If you invoke AGE from a standard Unix shell, any parentheses will have to be quoted or escaped.

For instance the argument **`tour`**`(`$[t]$`)` for tournament selection means that the tournament selection has one optional parameter t (the tournament size). Tournament selection can therefore be specified as `-S tour` to use the default tournament size, or as `-S tour(4)` to use a size of 4, in which case the command line options will usually have to be entered as `-S 'tour(4)'` or `-S tour\(4\)`.

Interface: The source file that contains the interface of the component. Recursively included header files are not listed. Additionally, all implemented algorithm elements can be including using the convenience header file `Elements.h`.

Implementation: The source files that contain the implementation of a component. The files the implementation depends on are listed only if they are not part of the AGE library.

6.3.1 Initialisers

RandomInitialiser implements random initialisation.

Argument: **`random`**`(`m–n`)`

m to n (inclusive) is the range of chromosome lengths.

Interface:	`RandomInitialiser.h`
Implementation:	`RandomInitialiser.cpp`

RampedInitialiser implements a generalisation of the "sensible" initialisation method devised by O'Neill and Ryan (2003, sec. 8.8), which in turn is based on Koza's ramped half-and-half initialisation. See Section 5.2 for a discussion.

In addition to the range of derivation tree heights, a number of other optional parameters are implemented: a specific "grow" rate (the default is one half initialised by the "grow" method, one half by the "full" method, hence the name "half-and-half"), which can be either stochastic (interpreted as probability, as in GEVA), or deterministic (interpreted as a ratio, as by Koza, 1992); a special treatment for recursive productions (discussed in Section 5.2); generation of unique trees (discussed in the same section); and generation of "tails" with random degenerate codons (as implemented in libGE: Nicolau and Slattery, 2006).

Argument: **`ramped`**`(`m–n`[`, *stoch*, *grow*, *oneill*, *u*, *tl*, *tr*`])`

m to *n* (inclusive) is the range of derivation tree heights, $m = 0$ is replaced by the lowest height possible for a given grammar;
stoch indicates whether the "grow" rate is to be interpreted as a probability (`1`), or a ratio (`0`, default);
grow is the "grow" rate from $[0, 1]$, default: 0.5;
oneill indicates whether recursive productions receive special treatment "O'Neill-style" (`1`), or not (`0`, default), see Section 5.2;
u indicates whether each generated tree is unique (`1`), or not (`0`, default);
tl is the absolute length of the tail of random codons (default: 0);
tr is the ratio of the tail length to the significant part length (default: 0).

At least one of *tl* and *tr* must be 0.

Interface:	`RampedInitialiser.h`
Implementation:	`RampedInitialiser.cpp`

6.3.2 Selectors

All implemented selection methods are *with replacement* in the sense that the same individual can be re-selected by repeated invocation of a given selection method. All tournament selection methods are *without replacement* within a given tournament (discussed in Section 2.6).

RouletteWheelSelector implements roulette-wheel selection as described in Section 2.6.

Argument:	**`roulette`** (no parameters)
Interface:	`Selectors.h`
Implementation:	`Selectors.cpp`

RandomTournamentSelector implements a variant of tournament selection. It is *without replacement* within a given tournament. Called simply "tournament" by Koza (1992). Similar to the implementation in GEVA, which is, however, *with replacement* within a given tournament.

Argument: **`tour`**`(`[t]`)`

t tournament size (default: 2).

Interface: `Selectors.h`

Implementation: `Selectors.cpp`

WetzelTournamentSelector implements a variant of tournament selection. It is *without replacement* within a given tournament, and picks competitors using roulette-wheel selection. It is the only tournament selection method, "stochastic tournament" or "Wetzel ranking", described by Goldberg (1989). Similar to the implementation in GAlib, which is however, *with replacement* within a given tournament, and only allows tournaments of size 2.

Argument: **`wtour`**`(`[t]`)`

t tournament size (default: 2).

Interface: `Selectors.h`

Implementation: `Selectors.cpp`

6.3.3 Fitness scalings

FitnessReversal implements a reversal transformation $f' = c - f$. The constant c is by default the maximum fitness in the current generation, or alternatively the sum of the maximum and the minimum. The former maps maximum to zero, the latter maps maximum to minimum and vice versa, and thus preserves the range of values.

Argument: **`rev`**`(`[*preserve*]`)`

preserve indicates whether to preserve the range of fitness values (`1`), or simply map the maximum to zero (`0`, default).

Interface: `FitnessScalings.h`

Implementation: `FitnessScalings.cpp`

AdjustedFitnessInversion implements Koza's adjusted fitness scaling.

Argument: **`adj`** (no parameters)

Interface: `FitnessScalings.h`

Implementation: `FitnessScalings.cpp`

FitnessInversion implements a simple inversion $f' = 1/f$, as suggested by Nicolau and Slattery (2006). Implemented primarily for comparison with Koza's adjusted fitness. Its usefulness may be limited.

Argument:	**`inv`** (no parameters)
Interface:	`FitnessScalings.h`
Implementation:	`FitnessScalings.cpp`

LinearFitnessScaling implements Goldberg's linear scaling with factor k.

Argument:	**`lin`**`(`[k]`)`

k is the factor k, expected number of copies of the best individual, $k \geq 1$, default: 1.2.

Interface:	`FitnessScalings.h`
Implementation:	`FitnessScalings.cpp`

6.3.4 Crossover operators

A probability of any crossover operator (incidentally only one is currently implemented) is set by the **`-x`** command line option, separately from other parameters.

OnePointCrossover implements one-point crossover on variable-length chromosomes, or alternatively on fixed-length chromosomes. If a chromosomes is at least two codons long, the crossover occurs at one of its inner points, not before the first codon or past the last codon.[1]

Argument:	**`one-point`**`(`[*fixed*]`)`

fixed indicates whether to preserve the length of chromosomes (`1`), or not (`0`, default).

Interface:	`Operators.h`
Implementation:	`Operators.cpp`

6.3.5 Mutation operators

A probability of mutation operators is set by the **`-m`** command line option (see Section 6.2), separately from other parameters.

BitMutation implements bit-level mutation using a faster algorithm. Recommended for general use instead of **`SlowBitMutation`**. See Section 5.4 for discussion and Section 7.2 for implementation details. Performance tests are done in Section 7.2 and in an experiment in Section 8.2.3.

Argument:	**`bit`** (no parameters)
Interface:	`Operators.h`
Implementation:	`Operators.cpp`

[1] This ensures that at least some genetic material, although possibly degenerate, is exchanged. The same method has been used by Goldberg (1989) and in libGE. In GEVA crossover can occur before the first codon at both chromosomes, which just swaps their contents.

SlowBitMutation implements bit-level mutation using a naïve, and significantly slower, algorithm. The implementation in **`BitMutation`** is recommend for general use, see discussion in Section 5.4.

Argument:	**`slowbit`** (no parameters)
Interface:	`Operators.h`
Implementation:	`Operators.cpp`

CodonMutation implements codon-level mutation exactly in the same way as it is implemented in GEVA.

Argument:	**`codon`** (no parameters)
Interface:	`Operators.h`
Implementation:	`Operators.cpp`

Duplication implements duplication according to its description by O'Neill and Ryan (2003).

Argument:	**`duplication`** (no parameters)
Interface:	`Operators.h`
Implementation:	`Operators.cpp`

6.3.6 Implemented applications

The fitness evaluators and the related application-dependent classes are implemented outside the AGE library and are used in the demonstration command line tool. To make the separation clear, the applications are implemented in files whose names begin with "`App-`".

RegressionFitnessEvaluator implements symbolic regression, used for experiments in Section 8.2.

Argument: `reg(`[*language*, *m*-*n*, *square*, *maxWraps*, *bnf*]`)`

language identifies the language for evaluation, either `lua` (default), `lua-safe` or `c`.

m to *n* (inclusive) is the range of exponents of the target polynomial $x^m + \cdots + x^n$, default: 1 to 4.

square indicates whether to square errors (`1`, default), or use an absolute value (`0`).

maxWraps is the maximum number of wrapping events, default: 3.

bnf is path to the BNF grammar file, default: the grammar used in Section 8.2.

Interface: `App-RegressionFitnessEvaluator.h`

Implementation: `App-RegressionFitnessEvaluator.cpp`, `App-Regression.h`, `App-Regression.cpp`

AntTrailFitnessEvaluator implements the ant trail problem, used for experiments in Section 8.3, implementation discussed in the tutorial for application developers in Section 6.6.

Argument: `ant(`[*higherbetter*`,` *time*`,` *trail*`,` *maxWraps*`,` *bnf*]`)`

higherbetter indicates whether standard fitness ordering is used (`1`) instead of reverse ordering (`0`, default).

time is the time limit, default: 600.

trail is the trail map file, default: Santa Fe ant trail.

maxWraps is the maximum number of wrapping events, default: 3.

bnf is path to the BNF grammar file, default: the grammar used in Section 8.3.

Interface: `App-AntTrailEvaluator.h`

Implementation: `App-AntTrailEvaluator.cpp`, `App-AntTrail.h`, `App-AntTrail.cpp`

6.4 File formats

AGE optionally saves a more detailed output in a directory specified by the **`-o`** option. The directory contains the following files:

- **`AGE.xml`**, which lists the command line options,
- **`EvolutionRun-`***nnnn***`.xml`** files, which include detailed data in XML from each run,
- **`AGE.xsl`**, which contains a XSL style sheet for visualisation of the above files,
- **`GenerationHits.txt`**, which contains cumulative numbers of fitness hits aggregated by generation, if a limit is specified by the **`-h`** option,
- **`GenerationStats.txt`**, which contains averaged overall statistics aggregated by generation.

As can be seen, the output consists of XML data and text. The two text files do not contain any information that could not be computed from the XML data, they are created for the user's convenience.

6.4.1 XML data

The **`AGE.xml`** file is a simple XML file consisting of a single element **`options`**, which contains the command line options supplied to AGE, one per line. The file is in XML to allow for future extension with more structured parameters while maintaining backwards compatibility.

The **`EvolutionRun-`***nnnn***`.xml`** files are generated for each run. Their names contain a zero-padded numbers of the runs counting from zero (*nnnn*). Their format consists of a hierarchy of elements:

`<evolution run="`*n*`" seed="`*s*`">` *populations stopped* `</evolution>`

is a top-level element that delimits the run number n of the evolutionary algorithm. The random number generator seed s is indicated so that the single run can be reproduced using the **`-r`** option (without **`-n`**). The element contains multiple **`population`** elements, one for each generation of the run, and one **`stopped`** element.

```
<population
   generation="g"  fitnessOrder="fo"  fitnessRange="fr"
   minFitness="minF"  maxFitness="maxF"
   avgFitness="avgF"  varFitness="varF"
   invalids="i"
   avgLength="avgL"
[  avgUsedCodons="avgUC"  avgWraps="avgW"
   avgDerivationTreeHeight="avgDTH" ] (optional)
   crossovers="c"  mutations="m"
>  individuals  </population>
```

delimits and describes a population of a generation g. The fitness ordering *fo* is either **`HigherBetter`** (standard), or **`LowerBetter`** (reverse), the fitness range *fr* is either **`Unbounded`** or **`Nonnegative`**. The fitness statistics (*minF*, *maxF*, *avgF*, *varF*) are computed from all valid individuals in the population.

The number of invalid individuals is i. The average length of chromosome in codons is *avgL*.

The average number of used codons (multiple use is counted) *avgUC*, the average number of wrapping events *avgW*, and the average derivation tree depth *avgDTH* are present only if made available by the fitness evaluator, and, again, are computed only from valid individuals.

The number of crossover operations is c, and the numbers of mutation operations are listed in m, a comma-separated list of counts for each mutation operator.

If individual information output is enabled by the **`-i`** option, the element contains **`individual`** elements, one for each individual in the population. Otherwise, it contains the **`noindividuals`** element.

`<stopped reason="r" />`

indicates why the evolutionary algorithm was stopped. The attribute r may have one of the following values:

– **`MaxGeneration`**, when the maximum number of generations was reached,

– **`Halted`**, when the algorithm was halted (due to an error in another run or for other external reason),

– **`StopLimit`**, when the fitness reached the limit specified by the **`-f`** option,

– **`Failed`**, when an error occurred.

```
<individual  fitness="f" description="d"
             [best="1"] [invalid="1"]       />
```

describes a single individual with a fitness value f and an application-dependent description *d*. The optional parameters denote an invalid individual or the best individual of the generation. Only one individual is labelled as the best one, even if there are more of them with an equal fitness value.

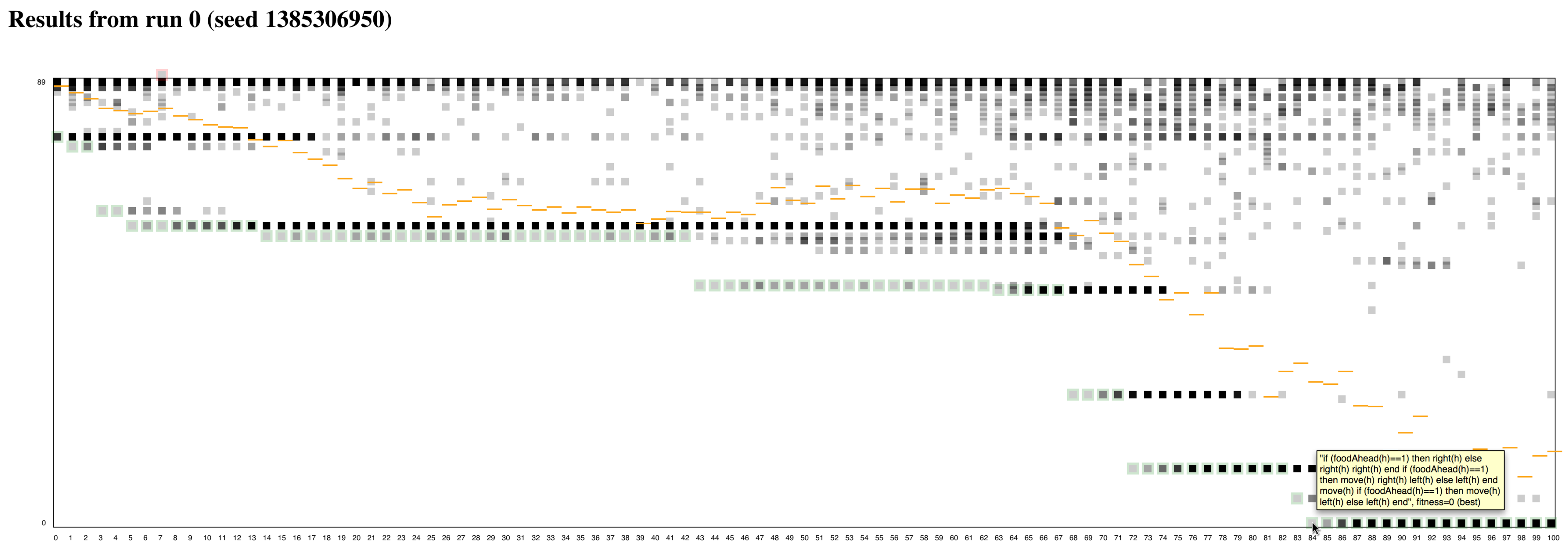
`<`**`noindividuals`**` size="`*n*`" />`

is a placeholder for *n* individuals if individual information output is not enabled by the **`-i`** option.

The **`AGE.xsl`** file contains a XSL style sheet that transforms the XML output into a CSS-styled HTML visualisation of the results suitable for viewing using a web browser. The transformation can be done either using a XSLT processor, such as `xsltproc`, or directly in a web browser that supports it. As of this writing, Mozilla Firefox 3.0 has a sufficiently powerful implementation of XSL to apply the style sheet.

In Figure 6.1 you can see results from a run of the Santa Fe ant trail experiment as displayed in a web browser. All individuals are shown in a graph and it is possible to view details about each of them.

While such a visualisation can certainly be useful, it also serves as a demonstration of how easily can the XML data produced by AGE processed given the wide support XML has gained across a wide range of software tools.


Results from run 0 (seed 1385306950)

"if (foodAhead(h)==1) then right(h) else right(h) right(h) end if (foodAhead(h)==1) then move(h) right(h) left(h) else left(h) end move(h) if (foodAhead(h)==1) then move(h) left(h) else left(h) end", fitness=0 (best)

Figure 6.1: Visualisation of XML data in a web browser using a XSL style sheet: a run of the Santa Fe ant trail application. Translucent boxes mark individuals, short orange lines mark averages, boxes with a red border mark invalid individuals (see the out-of-bounds individual in generation 7), boxes with a green border mark best-of-generation individuals. Hovering the mouse cursor over them displays the individual's description, fitness value and best/invalid information. (Colours not visible in black and white print.)

6.4.2 Text data

The text files consist of a one-line header and lines of data, one for each generation. The data are divided into columns with names specified in the header. Columns are separated by tab characters and may be padded to a certain width with spaces. The format of the two following files is also used for the corresponding sections of the console output.

The **`GenerationHits.txt`** file, which contains cumulative fitness hits if activated by the **`-h`** option, has two columns:

`Gen`
: contains the generation number.

`Runs with fitness better than or equal to` f
: contains the number of runs that reached fitness better than or equal to f, the fitness hit limit specified by the **`-h`** option.

The **`GenerationStats.txt`** file contains averaged statistics of all runs. The data is aggregated by generation, so that each line contains an average taken over all runs of statistics in a given generation. The file has the following columns, which correspond to attributes of the **`population`** element in the XML output:

`Gen`
: contains the generation number.

`Invalids`
: contains the average number of invalid individuals.

`BestFitness`
: contains the average of best fitness values.

`WorstFitness` (if verbose output enabled)
: contains the average of worst fitness values.

`AvgFitness`
: contains the average of average fitness values.

`VarFitness` (if verbose output enabled)
: contains the average of variances of fitness values.

`AvgLength` (if verbose output enabled)
: contains the average of average lengths of chromosome in codons,

`AvgUsdCodons` (if verbose output enabled)
: contains the average of average numbers of used codons (multiple use is counted).

`AvgWraps` (if verbose output enabled)
: contains the average of average numbers of wrapping events.

`AvgTrHeight` (if verbose output enabled)
: contains the average of average tree heights.

As in the XML output, invalid individuals are excluded from the statistics, and the last three columns are present only if the information is made available by the fitness evaluator. Verbose output can be enabled by the **`-v`** option.

6.5 Application programming interface

The application programming interfaces of AGE stem from its overall design, discussed in Section 5.1.

The EA/GE Engine API will be described in the following sections, each covering one or more header files:

- Section 6.5.1 covers basic data types, provided by `Base.h` and `yasper.h`.
- Section 6.5.2 covers error (exception) classes, provided by `Error.h`.
- Section 6.5.3 covers random number generation and related interfaces, provided by `Random.h`.
- Section 6.5.4 covers more advanced types and functions for working with fitness, provided by `FitnessUtils.h`.
- Section 6.5.5 covers individuals, data types that constitute individuals, and individual collections, provided by `Individual.h`, `Chromosome.h`, `CodonVector.h`, and `IndividualCollection.h`.
- Section 6.5.7 covers classes that implement the GE mapping, provided by `Grammar.h` and `BNFGrammar.h`.
- Section 6.5.6 covers interfaces implemented by component classes, provided by `Initialiser.h`, `Operator.h`, `FitnessEvaluator.h`, `FitnessScaling.h`, and `Selector.h`.

The Command Line Tool API is described in Section 6.5.8, and provided by `ArgUtils.h` and `Tool.h`.

The APIs are grouped by data types (abstract and concrete classes, structures, type definitions) and, in a few cases, global functions. If you are reading the electronic version, all data type names that occur throughout the text are hyperlinks to their descriptions. Description of each data type lists all of its public fields and methods relevant for application developers.

In the following text I will often say that an abstract class is, or defines, an *interface*, and I will refer to abstract classes as to interfaces. If a class *implements* an interface, it technically means that it inherits from an abstract base class and implements all of its pure virtual methods.

After building and installing AGE, the header files that provide the API are located in a subdirectory `AGE` of `/usr/local/include` or another `include` directory, based on the installation setup (see Section 6.1). All data types and global functions declared or defined in them are contained in the **`AGE`** namespace, including (`AGE::`)**`yasper::ptr`**. The inclusion of AGE headers also results in definition of several preprocessor macros, all of which either come from standard C and C++ header files or are prefixed with `AGE` or `_AGE`.

The API header files are covered by the same opensource licence as the whole software project, the three-clause BSD licence, with the exception of `yasper.h` covered by the similarly liberal zlib/libpng licence. See Section 6.7 for complete information.

6.5.1 Basic data types

`Base.h` provides several basic data types and one interface, technically a pure abstract class:

Codon is an unsigned integral data type with a sufficient range to hold codons. The number of bits per codon is defined by the constant **`CodonBits`**; the number of possible codon values is defined by the constant **`CodonValues`** (equal to 2^n, $n =$ **`CodonBits`**). The actual width of the **`Codon`** type may be larger, in that case only the lower-order **`CodonBits`** bits are taken into account, and the higher-order bits must be set to zero. The data type and the related constants are determined by the preprocessor macro `AGE_CODON_SIZE`, which can be used to set a custom codon size in bits.

Range is a structure that represents a range of unsigned integers, used in a variety of situations. It consists of two public fields and provides a convenience constructor:

`unsigned `**`start`**`;`
is the first valid value of the range,

`unsigned `**`size`**`;`
is the number of values in the range.

`Range``(unsigned aStart, unsigned aSize);`
creates a range of `aSize` values starting with `aStart`.

Fitness is a floating-point data type with a sufficient precision for representing raw or scaled fitness values. It is guaranteed to be at least as precise as `float`, the built-in single precision data type it currently defines.

Valid fitness values are finite, the minimum (smallest negative) and maximum (largest positive) values of fitness are accessible as the constants **`FitnessMin`** and **`FitnessMax`**. Values not in [**`FitnessMin`**, **`FitnessMax`**] are invalid and may be reserved for special purposes. The only such invalid value that has a defined meaning is the constant **`FitnessUnknown`**, which is used as a placeholder for unknown fitness values. When implementing a fitness evaluator and fitness scaling you are responsible for returning valid fitness values.

More advanced APIs related to fitness are discussed in Section 6.5.4.

DFitness is a floating-point data type for intermediate fitness calculations with increased precision.

Cloneable is an interface implemented by all component classes. It consists of a single pure virtual method:

`virtual Cloneable* `**`clone`**`() const = 0;`

A class that implements **`Cloneable`** must be able to clone its objects so that all their parameters (at least those passed to the constructor) are preserved, but no state information, or information related to other objects is copied to the clone. The **`clone`**`()` method usually can be implemented using the `new` operator and a constructor. The **`clone`**`()` method should be defined with a covariant return type in concrete subclasses. See Section 6.5.6 for interfaces that inherit from **`Cloneable`**.

Additionally, special pointer types are used for dynamically allocated objects:

yasper::ptr<*X*> is a smart, reference-counted pointer to type *X* provided by the yasper library (contained in the `yasper.h` header file). When instances of class *X* are to be allocated dynamically, its definition also contains *X*`::`**`Ptr`**, which is defined as **`yasper::ptr`**`<`*X*`>`. Additionally, a handful of classes not used in contexts where reference counting is needed define the type *X*`::`**`APtr`** as the standard C++ automatic pointer `std::auto_pointer<`*X*`>`.

6.5.2 Errors

Throughout the AGE library, errors are handled using C++ exceptions. The header `Error.h` provides two exception classes to this purpose: **`Error`** and **`User-Error`**, both of which subclass **`Cause`**. You can use these exception classes to report errors from component classes (see Section 6.5.6).

Error is a subclass of `std::runtime_error`. Use it for unrecoverable errors caused by unexpected conditions, such as shortage of resources.

`AGEError``(`*description*`,` *cause*`)`
`AGEErrorNoCause``(`*description*`)`

Use these preprocessor macros instead of constructors. The parameter *description* is an error description (an instance of `std::string`), and the parameter *cause* is a standard `errno` value or an instance of another **`Error`** or **`UserError`**.

UserError is a subclass of `std::logic_error`. Use it for errors caused by user setup or user-supplied data. Make sure that you always generate a clear, descriptive message (the parameter `description`) for user errors.

```
explicit
UserError( const std::string& description );
UserError( const std::string& description,
           const Cause&       cause       );
UserError( const std::string& description,
           int                anErrnoCause );
```

Use these constructors to create an exception, optionally with a cause, which is either a standard `errno` value or an instance of another **`Error`** or **`UserError`**.

Cause is an abstract base class for **`Error`** and **`UserError`**. If you use AGE in a custom executable, you should catch exceptions of these two classes. You can then use the method `fprint()` to write the exception to a specified stream, or the shorthand method `logAndFail()` to print the exception to the standard error stream and terminate with the standard `EXIT_FAILURE` status:

```
void  fprint(FILE* f)                    const throw();
void  logAndFail(const char* introMsg)  const throw();
```

where `introMsg` is an introductory message printed before the exception information itself in a similar fashion as by the standard function `perror()`.

6.5.3 Random numbers

All interfaces related to the generation of random numbers are in the header `Random.h`. The random number generator is implemented in the class **`Random`** and does not rely on any external library. The class **`BiasedCoin`** can be used to implement operators, which are applied with a specified probability.

Random::Seed is an unsigned integral data type for representing random number generator seeds, numbers used to initialise its state. It is defined to the built-in type `unsigned` and any unsigned value is accepted. It should be however noted that the value 0 is always interpreted as 1, zero should therefore be avoided in order to guarantee that each seed generates a different sequence.

Random implements a random number generator. Each instance of class **`Random`** maintains its own state, which makes it suitable for a multithreaded environment. Algorithm elements receive a reference to an instance of **`Random`** and must not use any other external or internal source of randomness.

```
unsigned  random();
```

generates a random integer from 0 to **`AGE_RANDOM_MAX`** (2^{31}).

```
double    randomLessThanOne();
```

generates a random floating-point number from the interval $[0, 1)$.

```
double    randomLessThan(double high);
```

generates a random floating-point number from the interval $[0, \texttt{high})$.

```
Codon     randomCodon();
```

generates a random codon value.

If you need random numbers in a fitness evaluator, you can create your own instance of **`Random`** using the following constructor:

```
explicit  Random(Seed seed);
```

creates a random number generator and initialises its state with the parameter `seed`. See **`Random::Seed`** above.

BiasedCoin can be used to simulate Bernoulli trials with a specified probability of success. The probability values are embedded in **`BiasedCoin`** instances, which are immutable, except for the assignment operator. The generator is supplied as a parameter to the **`flip()`** method. Operators should use this class unless they need a more elaborate interpretation of probability.

BiasedCoin();

creates an invalid biased coin (with unset probability); **isValid**() will return `false`. This default constructor can be used to create a placeholder instance until the probability is set and a valid biased coin is assigned.

explicit
BiasedCoin(float prob);

creates a valid biased coin with probability of success `prob` from $[0, 1]$; **isValid**() will return `true`.

bool **isValid**() const;

indicates whether the probability was set by the constructor.

bool **isZero**() const;

indicates whether the probability is exactly zero. Note that operators with zero probability are not applied, checks in the operator's implementation are therefore superfluous.

bool **flip**(Random& r) const;

simulates a Bernoulli trial using the supplied random number generator `r`, and indicates whether it was successful.

6.5.4 Fitness

`FitnessUtils.h` provides additional types and functions related to fitness, which form an infrastructure for flexible and correct handling of fitness as outlined in Section 2.2. Most notably, the fitness values are associated with a range (**FitnessRange**) and ordering (**FitnessOrder**).

FitnessVector is a shorthand for `std::vector<`**Fitness**`>`, and can be used interchangeably. Fitness vectors stand for the fitness of multiple individuals when other properties of the individuals are not significant, for instance when scaling fitness or applying selection.

FitnessOrder is an enumeration type that defines two orderings of fitness values:

- **FitnessOrderUnknown** is a special value for an unknown or undefined ordering,
- **LowerBetter** is the reverse ordering,
- **HigherBetter** is the standard ordering.

FitnessRange is an enumeration type that defines two ranges of fitness values:

- **FitnessRangeUnknown** is a special value for an unknown or undefined range,
- **FitnessUnbounded** specifies values from [**FitnessMin**, **FitnessMax**],
- **FitnessNonnegative** specifies values from [0, **FitnessMax**]

Note that there is no range of [**FitnessMin**, 0], as it would be equivalent to the **FitnessNonnegative** range with a reverse ordering.

The header file also provides an interface to static functions for simple operations with these data types:

```
FitnessVector  FitnessVectorFromDoubles(
                 FitnessRange r,  std::vector<double> dv );
```

converts a vector of values of type `double` to a vector of valid fitness values from range `r`. Values higher than **`FitnessMax`** are truncated. If the range is **`FitnessUnbounded`**, values lower than **`FitnessMin`** are also truncated. If the range is **`FitnessNonnegative`**, negative values are not accepted. Infinite values are truncated properly, but NaN values are not accepted.

```
FitnessVector  FitnessVectorFromInts(
                 FitnessRange r,  std::vector<int> iv );
```

converts a vector of values of type `int` to a vector of valid fitness values from range `r`. Values higher than **`FitnessMax`** are truncated. If the range is **`FitnessUnbounded`**, values lower than **`FitnessMin`** are also truncated. If the range is **`FitnessNonnegative`**, negative values are not accepted.

```
FitnessOrder   FitnessOrderOpposite(   FitnessOrder order );
```

returns the opposite order.

```
Fitness        FitnessBest(            FitnessRange range,
                                       FitnessOrder order );
Fitness        FitnessWorst            FitnessRange range,
                                       FitnessOrder order );
```

return the best and worst fitness values for a given range and ordering.

```
bool           FitnessIsBetterOrEqual( FitnessOrder order,
                                    Fitness f,  Fitness g );
```

returns `true` when `f` is better than or equal to `g` in the specified ordering, otherwise returns `false`.

FitnessStats is a structure for fitness statistics of multiple individuals. It consists of four public fields:

```
Fitness  min;
Fitness  max;
Fitness  avg;
Fitness  var;
```

that represent the minimum, maximum, average and variance of fitness values. Invalid individuals are not included in the statistics.

6.5.5 Individuals

Individuals, objects of class **`Individual`**, consist of genotype, represented by instances of the class **`Chromosome`**, and the information derived from it:

- raw and scaled fitness (**`Fitness`**): determined by a fitness evaluator and a fitness scaling,

- information whether the individual is valid: a fitness evaluator can declare an individual invalid if it cannot be evaluated,
- optionally information about the GE mapping process (the derivation of a phenotype string, a **`DerivationInfo`** structure) and the phenotype string itself (an instance of **`StringPtr`**).

Collections of individuals are stored in an **`IndividualCollection`**, which defines convenience methods for setting properties of multiple individuals at once. When only the genotype is manipulated, the strings of codons are represented by **`CodonVector`**.

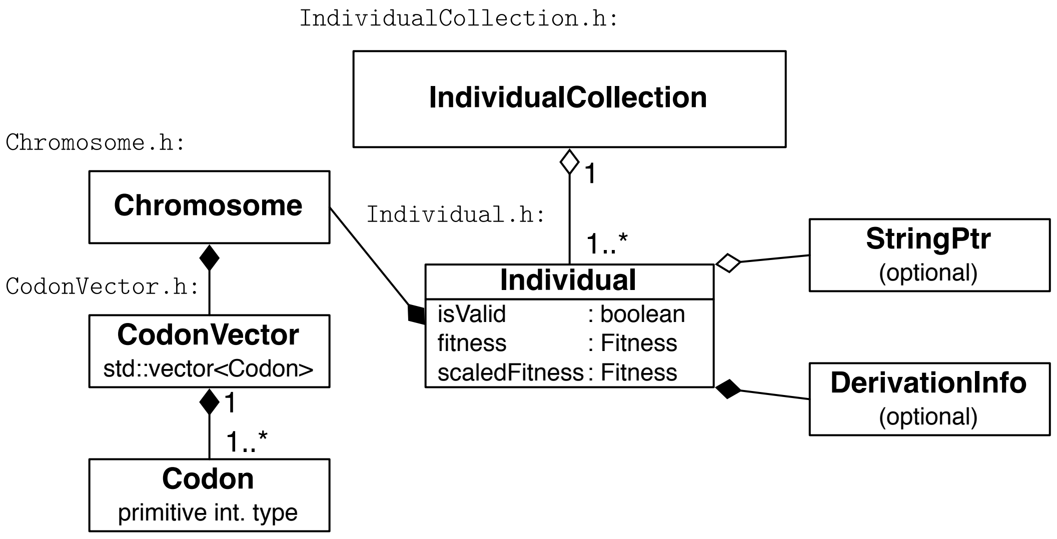


Figure 6.2: Individual-related classes.

Figure 6.2 presents the relationships between these classes and the basic data types along with the header files that provide them.

CodonVector is a shorthand for `std::vector<`**`Codon`**`>`, and can be used interchangeably. It is used to represent both fixed-length and variable-length codon strings. Operators are applied to **`CodonVector`**s.

Chromosome represents an individual genotype as a string of codons (a **`CodonVector`**). Note that, in contrast with the **`CodonVector`** itself, interface of this class allows only read access to the genotype, which is sufficient for a fitness evaluator or for output of individual information.

```
unsigned             size()        const;
```

returns the number of codons.

```
bool                 empty()       const;
```

indicates whether the codon string is empty.

```
const CodonVector&   codonVector() const;
```

returns a constant reference to the codon string.

DerivationInfo is a structure for storing information about the GE mapping, the derivation of a phenotype string. It consists of three public fields:

`unsigned wrapCount;`

is the number of wrapping events. If the maximum number m has been exceeded, it is set to $m + 1$.

`unsigned lastWrapCodonCount;`

is the number of codons used after the last wrapping event, or the total number of used codons if no wrapping event occurred.

`unsigned treeHeight;`

is the height of the derivation tree, the lowest valid value is therefore 1. If the maximum height h has been exceeded, it is set to $h + 1$.

The structure is usually obtained from the **`stringDerivedUsingCodons`**`()` method of a **`Grammar`** object (described in Section 6.5.7). The following constructors and methods are provided for convenience:

`DerivationInfo``();`

creates an invalid structure; **`isValid`**`()` will return `true`. This default constructor can be used to create a placeholder instance if the information is not available (for instance until an individual is evaluated).

`DerivationInfo``( unsigned w, unsigned c, unsigned h );`

creates a valid structure with **`wrapCount`** set to `w`, **`lastWrapCodonCount`** set to `c`, and **`treeHeight`** set to `h`. The derivation tree height `h` must be at least 1.

```
bool  wrapsExceeded(      unsigned maxWrapCount )  const;
bool  treeHeightExceeded(unsigned maxTreeHeight)  const;
```

are convenience methods for determining whether the maximum number of wrapping events or the maximum derivation tree height were exceeded. If the supplied maximum was also used as a constrain for the GE mapping (Section 6.5.7) and the method returns `false`, it means that the mapping process have failed, and the individual is therefore invalid.

```
bool  isValid()                                    const;
```

indicates whether the structure has been properly initialised.

StringPtr is shorthand for **`yasper::ptr`**`<std::string>`, a reference counted pointer to a dynamically allocated string. Dereferencing yields an instance of `std::string`. It is used for phenotype strings derived using the GE mapping, such as those returned by the **`stringDerivedUsingCodons`**`()` method of a **`Grammar`** object (described in Section 6.5.7).

Individual represents an individual. Instances contains all information related to a particular individual. You need to use this class only in a fitness evaluator, along with **`IndividualCollection`**. A fitness evaluator must set fitness to a valid value and validity to `true` or `false`. Invalid individuals should be assigned the worst fitness possible. An individual is usually declared invalid by the fitness evaluator when the GE mapping fails. An application can extend the notion of validity to other areas as well, or not use it at all. Invalid individuals are counted in each generation and excluded from the generation statistics. Consequently validity affects two areas:

– output, which contains both invalid individual counts,

– fitness scalings that depend on fitness statistics, such as Goldberg's linear scaling (**`LinearFitnessScaling`**).

The class defines the following methods for getting properties of an individual:

```
bool                    isValid()               const;
Fitness                 fitness()               const;
```

Until the validity and fitness values are assigned by the fitness evaluator, they are set to `false` and **`FitnessUnknown`**.

```
const Chromosome&       chromosome()            const;
const StringPtr&        phenotype()             const;
const DerivationInfo&   derivationInfo()        const;
```

The meaning of these methods follows directly from the description of the classes **`Chromosome`**, **`StringPtr`**, and **`DerivationInfo`**. When an application does not set phenotype, a `NULL` pointer is returned; when it does not set information about the GE mapping, an invalid **`DerivationInfo`** structure is returned. Note that only constant methods ("getters") are listed. This is because applications have only access to constant individuals, and set their properties using an **`IndividualCollection`**. Initialisers and operators, on the other hand, operate directly on **`CodonVector`**s.

IndividualCollection represents a collection of individuals, and defines convenience methods for setting properties of one or more individuals.

`iterator`
`const_iterator`

are iterators over the collection, and can be dereferenced to an **`Individual`** object. Only the constant iterator is available to applications.

```
const_iterator     begin()                 const;
const_iterator     end()                   const;
const Individual&  individual(unsigned i)  const;
unsigned           individualCount()       const;
```

can be used to access the individuals in the container, either using iterators or as an indexed array.

```
void  setFitnessForIndividual(        unsigned i,
                                       Fitness f  );
void  setValidityForIndividual(       unsigned i,
                                          bool v  );
void  setPhenotypeForIndividual(      unsigned i,
                              const StringPtr& ph );
void  setDerivationInfoForIndividual( unsigned i,
                         const DerivationInfo& di );
```

can be used by the fitness evaluator to set fitness (**`Fitness`**), validity, phenotype string (**`StringPtr`**), and information about the GE mapping (**`DerivationInfo`**) of an individual at index `i`.

```
void  setPropertiesForIndividual(     unsigned i,
       bool valid,  const StringPtr& phenotype,
       const DerivationInfo& derivationInfo       );
void  setFitnessForIndividualsWhereUnknown(
       const FitnessVector& fitness               );
```

are convenience methods for setting multiple properties for a single individual at index `i`, and for setting fitness (using a **`FitnessVector`**) for all individuals with unknown fitness. Size of the fitness vector must be equal to the number of individuals in the collection whose fitness is **`FitnessUnknown`**.

6.5.6 Components

The algorithm elements and applications in AGE are implemented as separate components. Interface for these components is defined in the form of abstract classes, from which the components should be derived. Figure 6.3 shows the hierarchy of these abstract classes, and the classes they are applied to. The diagram also provides a complete listings of methods the concrete classes need to implement. All methods are virtual, and with the exception of two methods that have a default implementation (**`FitnessEvaluator`**`::grammar()` and `description()`), they are pure virtual. The `virtual` keywords and the `=0` specifications have, therefore, been omitted from the following text for conciseness.

A component must implement an appropriate interface: one of **`Initialiser`**, **`CrossoverOperator`**, **`MutationOperator`**, **`FitnessEvaluator`**, **`FitnessScaling`**, and **`Selector`**. If you want to make your component class available to a command line tool built using the **`Tool`** class, you must also advertise its user-configurable parameters by implementing the informal **`ArgObject`** interface discussed in Section 6.5.8.

All interfaces inherit from **`Cloneable`** (see Section 6.5.1), which consequently must be implemented by all component classes. As a general rule, the **`clone`**`()` method must not copy anything else than the user-supplied parameters (passed to the constructor or as operator probabilities).

In addition to the abstract classes, the header `Operator.h` provides a convenience template for implementing operators **`CoinOperated`**<*O*>, which is also described in this section.

To report errors from component class methods, use the exception classes discussed in Section 6.5.2.

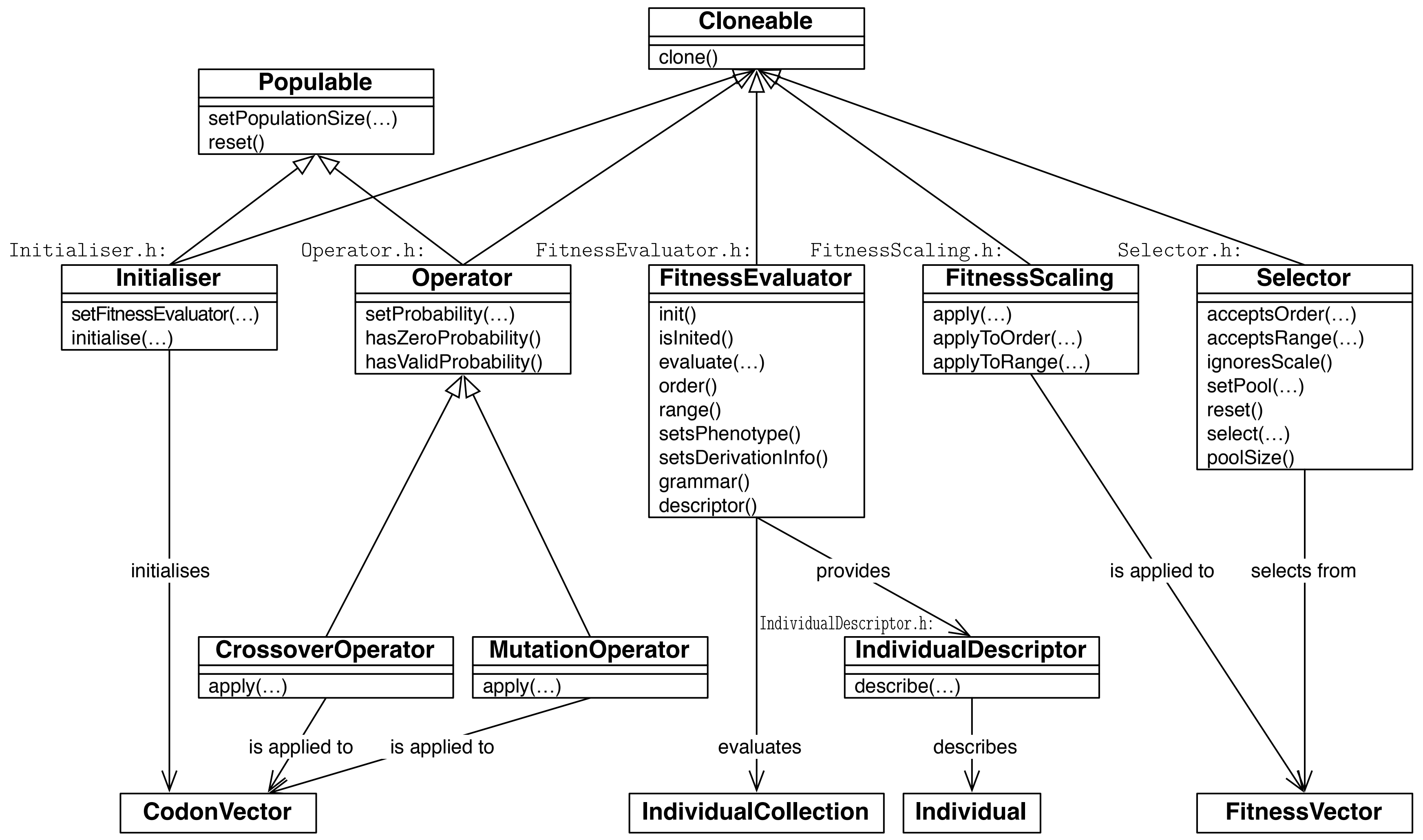


Figure 6.3: Abstract classes for algorithm elements and applications.

Populable defines two methods common to **`Initialiser`** and **`Operator`**. Both are applied to codon strings of single individuals or pairs of them to iteratively process a population. It is often desirable that the component can distinguish between applications to different populations. Even more importantly, when an operator or initialiser maintains internal state, it must not carry it over to a different population. If your operator or initialiser is stateless and you do not need such information, provide an empty implementation.

```
void  setPopulationSize(unsigned aSize);
```

is called before applying the algorithm element to the first individual of a population of size `aSize`. For crossover, `aSize` is the size of the new population. Instances of a concrete class can use the method to initialise their state, acquire resources, or prepare for a given number of individuals.

```
void  reset();
```

is called after applying the algorithm element to the last individual of the population. Any internal state must be discarded.

Initialiser inherits from **`Populable`**, and defines an interface for initialisation schemes.

```
void  setFitnessEvaluator(const FitnessEvaluator& fe);
```

is called at least once before the initialiser is used. An initialiser can retrieve whatever information it needs from the fitness evaluator. Unless called again before the initialiser is applied to another population (before calling **`setPopulationSize`**`()`), the initialiser must retain the information. The method is particularly useful for getting information about the grammar used by the evaluator.

```
void  initialise(  Random&      r,
                   CodonVector& chromosome) ;
```

is called sequentially for the number of individuals set by **`setPopulationSize`**`()`. The initialised codon string must have at least one codon.

Operator inherits from **`Populable`**, and is a common ancestor of **`CrossoverOperator`** and **`MutationOperator`**. It defines methods for setting and checking probability the operator is applied with:

```
void  setProbability( float p );
```

is called once before the operator is used. Unless called again before the operator is applied to another population (before calling **`setPopulationSize`**`()`), the operator must retain the probability. The probability must be also copied to a clone. The operator itself is responsible for interpreting the probability

```
bool  hasZeroProbability()       const;
```

indicates whether the operator has zero probability.

```
bool  hasValidProbability()     const;
```

indicates whether the probability has been set using **`setProbability`**`()`.

CoinOperated<*O*> is a template class that inherits from the class *O*, which should be either **`CrossoverOperator`** or **`MutationOperator`**. The template provides an implementation of probability methods specified by **`Operator`** using a **`BiasedCoin`**, and has empty implementations of the **`Populable`** methods. Subclasses can still override the **`Populable`** methods if needed, and use the following `protected` method to make decisions based on the operator's probability:

```
bool  flipCoin(Random& r)    const;
```

returns the result of **`BiasedCoin`**`::`**`flip`**`()`.

Most operators implemented in AGE (see Section 6.3.4 and 6.3.5) are implemented using this template, and can serve as examples.

CrossoverOperator inherits from **`Operator`** and defines an interface for crossover operators:

```
bool  apply(        Random& r,
        const CodonVector& parent0,
        const CodonVector& parent1,
              CodonVector* child0,
              CodonVector* child1  );
```

applies the operator to two parent codon strings, `parent0` and `parent1`, producing one or two offspring, pointed by `child0` and `child1`. If only one child is requested, a `NULL` is passed for `child1`. Any codons already present in the pointed children must be ignored and discarded.

The parents are guaranteed to be at least one codon long. The offspring produced by the operator must be at least one codon long.

The operator must be applied according to the random number generator `r`, an instance of **`Random`**, and the operator's probability. If it is effectively applied, the method returns `true`, otherwise it returns `false` and does not modify the children. The number of effectively performed operations is used for statistics.

MutationOperator inherits from **`Operator`** and defines an interface for mutation operators:

```
unsigned apply( Random&  r,
          CodonVector&  chromosome );
```

applies the operator to `chromosome`, a codon string.

The codon string is guaranteed to be at least one codon long, and must be at least one codon long after application of the operator.

The operator must be applied according to the random number generator `r`, an instance of **`Random`**, and the operator's probability. If it is effectively applied, the method returns a non-zero number of mutations, otherwise it returns zero and does not modify the codon string. It is up to the implementation how exactly the number of mutation is counted. The number of performed operations is used for statistics.

FitnessEvaluator defines an interface for fitness evaluators and provides a default implementation of two of its methods, **`grammar`**`()` and **`descriptor`**`()`. A fitness evaluator evaluates the fitness, and validity of individuals (instances of **`Individual`**), which are passed to it as a collection (an instance of **`IndividualCollection`**). Each evaluator is associated with an ordering and a range of fitness values. An evaluator can also, optionally, assign phenotype strings and information about the GE mapping to the individuals. An evaluator can provide a descriptor, an object used to obtain a human-readable description of individuals.

```
void          init();
```

is called before the evaluator is used for the first time, either after being constructed or cloned. It can be used to acquire resources and to initialise data structures.

```
bool          isInited()                    const;
```

indicates whether the evaluator has been initialised using **`init`**`()`.

```
void          evaluate(
                 IndividualCollection& population );
```

evaluates all fitness individuals in the collection (`population`) whose fitness is **`FitnessUnknown`**. It must assign a (raw) fitness value and validity to each of the evaluated individuals using the appropriate methods of the **`IndividualCollection`** class. It can also, optionally, assign phenotype strings (dynamically allocated as **`StringPtr`**) and information about the GE mapping (**`DerivationInformation`**) to the individuals. In that case, it must do so consistently and advertise it using the **`setsPhenotype`**`()` and **`setsDerivationInfo`**`()` methods. A `NULL` phenotype string can be assigned to an invalid individual.

Individuals are passed as a collection instead of individually to allow for optimisation. Each individual should, however, be evaluated independently from other individuals in the collection.

The **`evaluate`**`()` method should throw **`Error`**s only when absolutely necessary. Invalid individuals should not result in errors. Shortage of resources should, unless it could have already been discovered in the **`init`**`()` method.

```
FitnessOrder  order()                       const;
FitnessRange  range()                       const;
bool          setsPhenotype()               const;
bool          setsDerivationInfo()          const;
```

are used to advertise basic information about the evaluator: ordering and range of its fitness value, and whether it assigns phenotype strings and information about the GE mapping to individuals.

The following two methods have a default implementation, concrete subclasses can override them:

```
const Grammar::Ptr&              grammar()     const;
```

allows GE fitness evaluators to return their grammar. The returned grammar is currently used by the **RampedInitilisier**, but may be used for different purposes. The default implementation returns a `NULL` pointer, and is intended only for evaluators that do not use a grammar to perform the genotype-phenotype mapping. See **Grammar** in Section 6.5.7.

```
const IndividualDescriptor::Ptr&  descriptor()  const;
```

allows fitness evaluators to provide a descriptor for individuals. The default implementation returns a descriptor for hexadecimal representation of the codon string. See **IndividualDescriptor** for more details.

Example: Section 6.6 gives a tutorial on implementing a fitness evaluator.

IndividualDescriptor defines an interface for obtaining a description of an individual, its human-readable representation used in the output. Descriptors are not used as component classes on their own: instead, they are associated with **FitnessEvaluator**s by means of their **descriptor**() methods. Their interface consists of a single method:

```
std::string  describe(
                 const Individual& individual )  const;
```

returns a human-readable representation of the individual.

FitnessScaling defines an interface for fitness scalings. A fitness scaling can be limited to specific orderings and ranges of fitness values, and it can change the ordering and range of the values it is applied to, as outlined in Section 2.2. Additionally it can use the supplied statistics to adjust its parameters. A fitness scaling must be an immutable, stateless object.

```
void  apply( FitnessVector& fitness,
               FitnessOrder order,
               FitnessRange range,
         const FitnessStats& stats   )  const;
```

applies the scaling to the fitness values in the vector, which have the specified ordering and range (**FitnessOrder** and **FitnessRange**), and which correspond to a population. The fitness statistics (**FitnessStats**) can be used to adjust the parameters of the scaling. Note that the statistics are taken only from valid individuals, while fitness values of all individuals are scaled. This is intentional, as the fitness values of invalid individuals would skew the statistics. If the scaling needs, for some reason, the actual minimum, maximum, average or variance of all values, it should compute it from the `fitness` vector.

Upon return the `fitness` vector contains the scaled values. The values must be valid and must be within the range returned by **`applyToRange`**`()`, their relative order should remain unchanged, except when **`applyToRange`**`()` returns the opposite order, in which case their relative order should be reversed. The size of the vector must not be changed.

```
FitnessOrder  applyToOrder(FitnessOrder order)  const;
FitnessRange  applyToRange(FitnessRange range)  const;
```

return the order and range resulting from the application of the scaling to the order and range specified by the parameters. When a given order or range is not accepted by the scaling, the methods must return **`FitnessOrderUnknown`** or **`FitnessRangeUnknown`** to indicate it and prevent application to the fitness values.

Selector defines an interface for selection schemes. A selector does not work with individuals, instead it selects elements from a vector of fitness values, a *pool*. A selector can be limited to specific orderings and ranges of fitness values, and can maintain internal state until it is attached to a different pool.

```
bool      acceptsOrder( FitnessOrder fo )  const;
bool      acceptsRange( FitnessRange fr )  const;
bool      ignoresScale()                   const;
```

indicate whether the selector accepts fitness values of a given ordering and range, and whether it is concerned only about the relative order of fitness values, and thus ignores their scale.

```
void  setPool( const FitnessVector&  fv,
                     FitnessOrder  fo,
                     FitnessRange  fr );
```

is called before a sequence of selections from a given pool (**`FitnessVector`**) of fitness values, which have the specified ordering and range (**`FitnessOrder`** and **`FitnessRange`**). The number of subsequent selections may be lower or higher than the size of the pool, which is guaranteed to be at least one.

```
void      reset();
```

is called after a sequence of selections from a pool. Implementation must ensure that no state is carried over to another pool.

```
unsigned  select(Random& r);
```

performs selection from the previously set pool using the random number generator `r`, an instance of **`Random`**. Returns a valid zero-based index in the pool.

```
unsigned  poolSize()  const;
```

returns zero if the pool has not been set or has been reset, otherwise returns the size of the pool.

6.5.7 Grammatical evolution

In the previous sections we have already covered several classes that contain support for grammatical evolution:

- the **`Individual`** and **`IndividualCollection`** classes, which define accessors for a phenotype string (represented by a **`StringPtr`**) and information about the GE mapping (represented by **`DerivationInfo`**),

- the **`FitnessEvaluator`** interface, which includes the methods **`setsPhenotype`**`()`, **`setsDerivationInfo`**`()`, and **`grammar`**`()`.

If your fitness evaluator is based on grammatical evolution, you should use these APIs to provide information about the GE mapping. The mapping itself is provided by the class **`BNFGrammar`**, which implements the **`Grammar`** interface. The grammar objects are always allocated dynamically and accessed using the **`Grammar`** interface, which makes it possible to replace the **`BNFGrammar`** class with a class that implements a different genotype-phenotype mapping using a grammar.

The interfaces are defined in `Grammar.h`. The only implementation accessible from `BNFGrammar.h`.

Optionally, a fitness evaluator can use the abstract and concrete classes for generation and evaluation of source code in Lua and C. The interfaces are defined in the headers `GrammaticalCodeGenerator.h`, `GrammaticalCodeGenerator.h`, and `CodeLauncher.h`. The concrete classes for C are accessible from the headers `CGrammaticalCodeGenerator.h` and `CCodeLauncher.h`. The concrete classes for Lua are accessible from the headers`LuaGrammaticalCodeGenerator.h` and `LuaCodeLauncher.h`. Section 6.6 gives a tutorial on these APIs.

Grammar is an interface for classes that perform a mapping from genotype (**`CodonVector`**) to phenotype (**`StringPtr`**) using a grammar. The interface consists of three pure virtual methods:

```
StringPtr              stringDerivedUsingCodons(
                         const CodonVector&       codons,
                         unsigned              maxWraps,
                         unsigned          maxTreeHeight,
                         DerivationInfo*  derivationInfo
                                                ) const;
```

attempts to map the codon string codons to phenotype and returns a dynamically allocated phenotype string, or a `NULL` pointer if the mapping fails. The derivation is constrained by the maximum number of wrapping events `maxWraps` and the maximum height of the derivation tree `maxTreeHeight`. Information about the mapping of the supplied codon string is returned using the `derivationInfo` pointer in the form of a **`DerivationInfo`** structure. If you do not need that information, pass `NULL` as `derivationInfo`. Maximum derivation tree height can be any positive number, or the constant **`DepthInfinite`**, in which case the height is not limited. If either `maxWraps` or `maxTreeHeight` is exceeded, the mapping fails, and a `NULL` pointer is returned.

```
CodonDerivation::Ptr  randomDerivation(
                           Random&                        r,
                           unsigned   desiredMinTreeHeight,
                           unsigned          maxTreeHeight
                                                   ) const;
```

attempts to generate a random derivation with the desired minimum tree height and maximum tree height using a random number generator. The minimum indicates only a preference, while the maximum is a hard limit. If there is no desired minimum, pass 0; if production rules that can be used to derive a tree of arbitrary height are to be preferred, pass the constant **`DepthInfinite`**. The result is returned as a dynamically allocated object of a class that implements the **`CodonDerivation`** interface, which also defines the semantics of the return value. This method is currently used only by the **`RampedInitilisier`** component class.

```
unsigned              minimumTreeHeight()          const;
```

is the minimum tree height of a derivation. It is therefore the lowest possible `maxTreeHeight` argument value for the **`randomDerivation`**`()` method.

```
std::string           description()                const;
```

returns a human-readable representation of the underlying grammar.

GrammarChoiceVector is a vector of integers of arbitrary length that represents a derivation tree. It is defined as `std::vector<`**`GrammarChoice`**`>`, where **`GrammarChoice`** is defined as **`Codon`** (although its values do not need to correspond to codon values in any way). Each derivation tree should be represented by a different **`GrammarChoiceVector`**. Vectors are different if they differ in length or in value at any particular position.

CodonDerivation is an interface for retrieving information about a derivation. The **`randomDerivation`**`()` method of the **`Grammar`** interface returns its result as a dynamically allocated object of a class that implements this interface. The interface consists of three methods:

```
bool                          wasSuccessful()      const;
```

indicates whether a derivation with given parameters was successfully found.

```
const CodonVector&            derivationCodons()   const;
```

returns a codon string that results in a derivation with given parameters.

```
const GrammarChoiceVector&  derivationChoices()  const;
```

returns a unique representation of the derivation tree in the form of a **`GrammarChoiceVector`**.

BNFGrammar is currently the only implementation of the **`Grammar`** interface. Its **`stringDerivedUsingCodons`**`()` method performs the genotype-phenotype mapping defined for grammatical evolution by O'Neill and Ryan

(2003) (see Section 2.4). Its **randomDerivation**() constructs such a random derivation that can be used by **RampedInitialiser** (see Section 5.2), and consequently for the "sensible" initialisation method as described by O'Neill and Ryan (2003).

The mapping is carried out using a context-free grammar in Backus-Naur form, which is passed to the constructor:

```
explicit  BNFGrammar(const char*  bnfCString);
```

constructs a grammar from its specification in Backus-Naur form represented as a C string. Lines, including the trailing one, must be terminated with line feeds. The first nonterminal is interpreted as the start nonterminal. Literals (terminals) can be both unquoted and enclosed in double quotes. The two forms can be mixed unless they appear next to each other. Apart from the standard BNF syntax, single-line comments prefixed with `#` and multi-line comments enclosed in `/*` and `*/` are supported. Both kinds of comments must not be preceded by non-white characters on the line where they begin.

6.5.8 Command line tool

The component classes can advertise their user-configurable parameters by implementing the **ArgObject** informal interface, which is available in `ArgUtils.h` along with the related classes. This is necessary for the component to be accessible from the command line interface. The interface also separates the components from the command line interface.

The command line interface itself is implemented by the **Tool** class, whose interface is in `Tool.h`. The class provides several ways of customising the command line tool, most importantly through addition of components that implement the **ArgObject** interface.

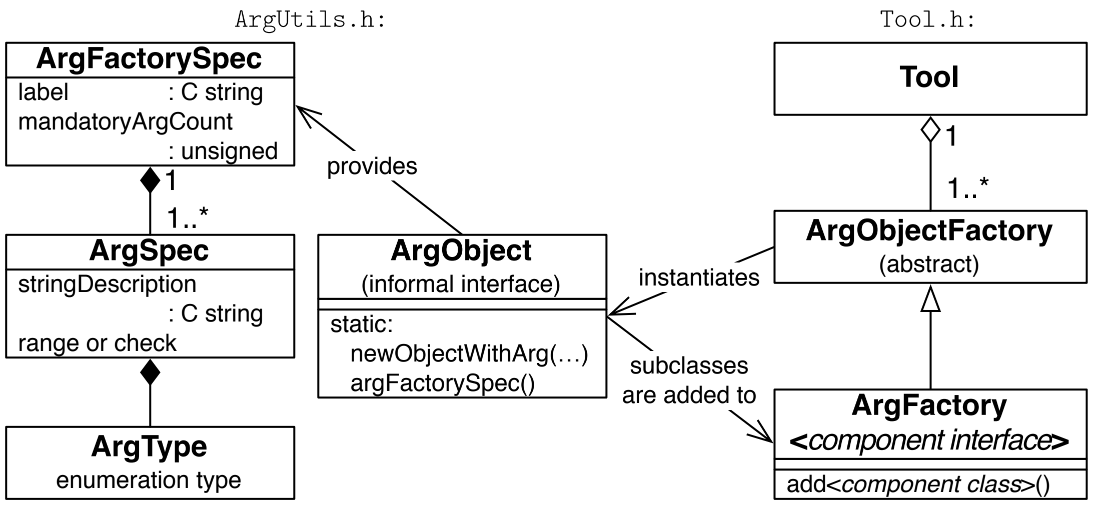


Figure 6.4: Relationships and interactions between the command line tool related classes.

Interactions between the **Tool**, a component class that implements **ArgObject**, and other related classes are shown in Figure 6.4. The **Tool** parses the command line arguments and instantiates components with appropriate parameters. To achieve this, a component classes must provide two methods: one that returns a specification of the component's parameters and the corresponding command line arguments, and one that instantiates a component according to the arguments. These methods constitute the informal **`ArgObject`** interface. We call it informal, as the C++ language does not have class objects, and consequently does not support virtual class methods.

ArgObject is an informal interface of two static methods. The component classes that implement it should by convention inherit from **`ArgObject`**, which is defined as an empty class with an empty virtual destructor.

```
static const ArgFactorySpec& argFactorySpec();
```

returns a specification of the user-configurable parameters of the component and the corresponding command line arguments in the form of an **`ArgFactorySpec`** structure.

```
static X* newObjectWithArg(
            const std::vector<ArgValue>& argv );
```

returns a new instance of the component. *X* is the interface implemented by the component, one of **`Initialiser`**, **`MutationOperator`**, **`CrossoverOperator`**, **`FitnessEvaluator`**, **`FitnessScaling`**, and **`Selector`** (see Section 6.5.6). The vector of parsed arguments (of type **`ArgValue`**) passed to this method complies with the specification returned by **`argFactorySpec`**`()`.

The method may throw a **`UserError`** if the arguments are incorrect in a way that does not follow from their specification, for instance if some of them have mutually exclusive values.

ArgType is an enumeration type that defines five types of arguments that can be passed to a concrete class that implements **`ArgObject`** to instantiate an object:

- **`UIntArg`** for an `unsigned`,
- **`FloatArg`** for a `float`,
- **`RangeArg`** for a **`Range`**, defined in Section 6.5.1,
- **`BoolArg`** for a `bool`,
- **`StringArg`** for a C string, pointed by a `const char*` pointer.

ArgSpec is a structure that specifies a single argument that can be passed to a concrete class that implements **`ArgObject`** to instantiate an object. Arguments have a human-readable label, a type (**`ArgType`**), and either a range of valid values (for numeric types) or a string checking function pointer (for strings) of the following type:

```
typedef bool (*StringCheck)(const char* str);
```

The return value of the pointed function indicates whether a given string is accepted. (Defined inside the **`ArgSpec`** definition.)

The structure consists of six fields, the last four of which are grouped in a `union`:

`const char*` **`label`**`;`

is a short human-readable description. Argument labels are printed in the help for a command line tool, where they are immediately followed by their range of valid values, with the exception of string labels. By convention, labels for optional arguments are suffixed with the default value in parenthesis, labels for string arguments are suffixed with a colon, space, and a specification of possible values. The label can be `NULL`, in which case only a type identifier is printed in the help.

Examples labels for numeric arguments: `"factor(1.2)"`, `"tail length(0)"`.

Example labels for string arguments: `"lang(lua): c|lua"`, `"bnf: path"`.

`ArgType` **`type`**`;`

is the type of the specified argument, according to which the following union is interpreted:

```
union{
  struct{  unsigned  from;
           unsigned  upto;  }  uintRange;
  struct{  float     from;
           float     upto;  }  floatRange;
  struct{  unsigned  from;
           unsigned  upto;  }  rangeRange;
  StringCheck                  stringCheck;
};
```

is, based on **`type`**, a range of valid `unsigned` values, or a range of valid `float` values, or a range of valid **`Range`** values, or a pointer to a string checking function. If the type is **`BoolArg`**, all fields are ignored. The ranges are inclusive, and may be effectively unbounded if the maximum or minimum value of a given type is supplied. The string checking function pointer may be `NULL` if a string parameter does not need to be checked for validity.

When an argument supplied from the command line is interpreted by AGE, it is parsed according to its type; if it is numeric, it is then checked against the range of valid values; if it is a string, it is then checked using the string checking function; finally, it is stored in an **`ArgValue`** and, together with other arguments, passed to the **`newObjectWithArg`**`()` method of a class that implements **`ArgObject`**.

ArgSpecVector is a shorthand for `std::vector<`**`ArgSpec`**`>`, and can be used interchangeably. It forms a part of the **`ArgFactorySpec`** structure.

ArgValue is a structure for storing argument values specified by **`ArgSpec`**. It consists of five mutually exclusive fields:

```
union{  unsigned  uintValue;
        float     floatValue;
        bool      boolValue;
        Range     rangeValue;  };
std::string       stringValue;
```

which are always interpreted in the context of an argument type, which determines the applicable field. (The `stringValue` field is excluded from the union because its type has a constructor. The `rangeValue` field is actually of a type different from `Range` defined in Section 6.5.1, but can be assigned `Range` values and converted to `Range`.)

ArgFactorySpec is a structure that specifies a *factory* for objects of a concrete class that implements `ArgObject`. Each component class accessible from a command line has its factory defined by the three fields of this structure:

```
const char*   label;
```

is a mandatory label for the argument of the factory. It should be a short identifier of the component class, unique within a given component type. The label is entered by the user to identify a component. The value "none" is reserved.

Example: The bit "bit" label identifies the `BitLevelMutation` class, the "codon" label identifies the `BitLevelMutation` class, the "one-point" label identifies the `OnePointCrossover` class. See Section 6.3 for labels of other components already implemented in AGE.

```
unsigned      mandatoryArgCount;
```

is the number of mandatory arguments. It must be less than or equal to the size of the `argSpecs` vector.

```
ArgSpecVector argSpecs;
```

is a vector of `ArgSpec` structures, one for each argument.

A structure of this type is returned by the `argFactorySpec()` method of a class that implements `ArgObject`.

Example: Listing 6.4 shows how the structure is constructed by the component class for Goldberg's linear scaling, `LinearFitnessScaling`.

Section 6.3 describes the forms of arguments for all implemented components from the user's point of view and explains the handling of mandatory and optional arguments. The constructions of the corresponding `ArgFactorySpec` structures can be found in the source code of the components.

ArgObjectFactory is an abstract superclass of instances of the `ArgFactory`<*X*> template. Instead of subclassing `ArgObjectFactory`, specialise the `ArgFactory`<*X*> template.

ArgFactory$<X>$ is a template class, a subclass of `ArgObjectFactory`. It is used to create a factory for components of a given type X, one of `Initialiser`, `CrossoverOperator`, `MutationOperator`, `FitnessEvaluator`, `FitnessScaling`, and `Selector`. The template class defines a default constructor and a single method template:

```
const ArgFactorySpec&  LinearFitnessScaling :: argFactorySpec(){
  static ArgFactorySpec spec = {  /*label:*/    "lin",
                                  /*mandatoryArgCount:*/ 0,
                                  /*argSpecs:*/ ArgSpecVector() };
  if( spec.argSpecs.empty() ){
    ArgSpec  as;
    as.label            = "factor(1.2)";
    as.type             = FloatArg;
    as.floatRange.from  = 1.0F;
    as.floatRange.upto  = FLT_MAX;
    spec.argSpecs.push_back(as);
  }
  return spec;
}
```

Listing 6.4: Example of **`ArgFactorySpec`** construction in an **`argFactorySpec`**`()` method. One optional argument, labelled `"factor(1.2)"`, is specified.

`ArgFactory``<`*X*`>();`

is the default constructor. The factory is actually constructed by the subsequent calls of **`add`**`<`*Y*`>()`.

`void` **`add`**`<`*Y*`>();`

adds a concrete class *Y* that implements **`ArgObject`** to the factory. Conformance to the informal interface is enforced at compile time.

If two components with the same **`ArgFactorySpec`** label are added, the method throws an **`Error`**.

Instances of **`ArgFactory`**`<`*X*`>` are used when constructing a **`Tool`** object. The classes added to the factories are then accessible from the command line tool based on the **`Tool`** object.

Example: Listing 6.5 includes the construction of a factory for each of the component types.

Tool is used to build a command line tool with the user interface described in Section 6.2. The component classes that will be accessible from the command line tool are added to factories (see **`ArgFactory`**`<`*X*`>`), the factories for each component type are then passed to the constructor of **`Tool`**, together with default arguments for each factory. The **`Tool`** object then takes over, processes command line arguments, runs evolutionary algorithms, and outputs results.

The class defines a constructor and three methods:

```
Tool( const ArgFactory<CrossoverOperator>& crossovers,
      const ArgFactory<MutationOperator>&   mutations,
      const ArgFactory<Initialiser>&     initialisers,
      const ArgFactory<FitnessEvaluator>&  evaluators,
      const ArgFactory<FitnessScaling>&      scalings,
      const ArgFactory<Selector>&           selectors,
      const char*                       defaultXvrArg,
      const char*                       defaultMtnArg,
      const char*                      defaultInitArg,
      const char*                      defaultEvalArg,
      const char*                     defaultScaleArg,
      const char*                       defaultSelArg );
```

constructs a **Tool** object with given factories and default arguments. The default arguments are exactly in the form in which they would be entered from the command line as described in Section 6.3, except that they do not contain the option prefix (**-X**, **-M**, **-I**, **-A**, **-s**, or **-S**). It is also possible to specify an empty string ("") as a default argument for mutations and fitness scalings, `defaultMtnArg` and `defaultScaleArg`. Any of the default arguments may be `NULL`, in which case there is no default argument, and the corresponding option is then mandatory.

The constructor throws an **Error** if any of the default arguments is invalid.

```
void  runOrFailWithArgs(
        int argc, char * const argv[] )  const;
```

runs the tool with given command line arguments. You can either directly pass the standard `argc` and `argv` arguments of the `main()` function, or preprocess them as needed.

The method may call `exit()`, throw an **Error** or a **UserError**.

```
void  fprintUsage(FILE* file)            const;
```

prints help to a given file descriptor.

```
void  logUsageAndFail()                  const;
```

prints help to the standard error stream, and then terminates the program with the standard `EXIT_FAILURE` status.

Example: Listing 6.5 shows a simple `main()` function built around a **Tool** object. The `main()` in the `main.c` source of the demonstration command line tool is also implemented using **Tool**.

```
#include "App-RegressionFitnessEvaluator.h"          /* application */
#include "Elements.h"                         /* algorithm elements */
#include "Tool.h"
#include "Error.h"                               /* Error, UserError */

using namespace AGE;

int  main(int argc, char * const argv[]){
 try{                     /* try and catch(Error), catch(UserError) */

    ArgFactory<CrossoverOperator>  crossoverFactory;
    crossoverFactory.add<OnePointCrossover>();

    ArgFactory<MutationOperator>   mutationFactory;
    mutationFactory.add<BitMutation>();

    ArgFactory<Initialiser>        initFactory;
    initFactory.add<RandomInitialiser>();
    initFactory.add<RampedInitialiser>();

    ArgFactory<FitnessEvaluator>   evalFactory;
    evalFactory.add<RegressionFitnessEvaluator>();

    ArgFactory<FitnessScaling>     scalingsFactory;
    evalFactory.add<LinearFitnessScaling>();

    ArgFactory<Selector>           selectorFactory;
    selectorFactory.add<RouletteWheelSelector>();

    Tool( crossoverFactory,  mutationFactory, initFactory,
          evalFactory,  scalingsFactory,  selectorFactory,
          "one-point",       "",              NULL,
          "reg",       "lin(1.5)",       "roulette"
        ).runOrFailWithArgs(argc, argv);

  }catch(const UserError& failCause){
    failCause.logAndFail("AGE failed due to the following error");
  }catch(const Error& failCause){
    failCause.logAndFail(
      "AGE failed due to the following internal error" );
  }                       /* try and catch(Error), catch(UserError) */
  return 0;
}
```

Listing 6.5: Example `main()` function of a command line tool built using the **`Tool`** class. The default arguments are `one-point` for crossover, no mutations, `reg` for application, `lin(1.5)` for scalings, `roulette` for selector. No default argument (`NULL`) is supplied for initialiser: the corresponding command line option will be mandatory. Any errors that could occur are caught and handled.

6.6 Tutorial for application developers

In this section, we will implement the Santa Fe ant trail application (used for experiments in Section 8.3) using the APIs available in AGE. Additionally, we will employ the helper classes **`GrammaticalCodeGenerator`**, **`LuaGrammatical-CodeGenerator`**, and **`LuaCodeLauncher`**, which facilitate the evaluation in Lua. These classes, and there counterparts for the C language, are part of the AGE library, but their API is defined only informally in this tutorial. Although their interfaces are simple, it is best understood how to use them from an example. Additional documentation is provided in comments in their header files named *classname*`.h`.

Before writing the fitness evaluator, we need to implement a *model class* that will encapsulate the domain-specific data and procedures. This is not the only way to build an application, but it makes for a better design and allows us to separate the part of implementation that can be written without the use of the AGE APIs.

Model class

The implementation of our model class, named `AntTrail`, follows directly from the description of the ant trail application in Section 8.3. Namely, it has the following methods:

```
void      left();
void      right();
void      move();
bool      foodAhead()  const;
unsigned  foodLeft()   const;
unsigned  foodEaten()  const;
unsigned  timeLeft()   const;
void      reset();
```

that correspond to the ant's actions, indicate the number of food pieces eaten and left, and reset the model to its initial state. Such a class can be implemented without any knowledge of the AGE APIs, and we will therefore leave its implementation as an exercise to the reader. The implementation used for experiments in Section 8.3 can be found in the files `App-AntTrail.h` and `App-AntTrail.cpp`, which are part of the AGE demonstration tool.

As the concrete subclasses of **`FitnessEvalautor`** must support cloning, the model class must also support either cloning or copying. In this case we will opt for cloning and dynamic allocation using a factory method:

```
static AntTrail*  newAntTrail( const char* mapCStr,
                               unsigned    time    );
AntTrail*         clone() const;
```

The factory method takes two arguments: the ant trail map as a C string, and a time limit. We will also define a constant that contains a C string representation of the standard Santa Fe ant trail map: `AntTrail::santaFeMap`.

As we want to evaluate individuals in the Lua language, the next step is to prepare the model class to be accessible from Lua. Another option would be to implement the whole model in the target language, but Lua makes it relatively easy to interface with C functions. Additionally, we will benefit from faster evaluation of compiled code.

Because Lua cannot work directly with C++ objects, we will need to write static equivalents of the above methods. As our model class is dynamically allocated, we can assign a unique identifier to each instance and provide a mapping from the identifiers to instance pointers. The identifier of a model object can be retrieved and then passed as an argument to the static methods, which will call an equivalent method on the designated object:

```
int          handle()  const;  /* a unique identifier */
static void  left(      int handle );
static void  right(     int handle );
static void  move(      int handle );
static int   foodAhead( int handle );
static int   foodLeft(  int handle );
static int   foodEaten( int handle );
static int   timeLeft(  int handle );
static void  reset(     int handle );
```

We have also replaced the `unsigned` type with `int`, as unsigned integers are not supported in Lua. Almost any scripting language with a C binding would require these changes.

Fitness evaluator and the helper classes

Now that we are done with the model class, let's proceed to the fitness evaluator. In addition to the **`FitnessEvaluator`** interface described in Section 6.5.6, it will also implement the informal **`ArgObject`** interface described Section 6.5.8, so that it is accessible from the command line tool.

We will subclass the **`GrammaticalCodeFitnessEvaluator`** abstract class (defined in `GrammaticalCodeFitnessEvaluator.h`), which implements several **`FitnessEvaluator`** methods using a **`GrammaticalCodeGenerator`** and a **`CodeLauncher`**.

A **`GrammaticalCodeGenerator`** maps individuals to phenotype using a grammar and generates source code in a certain language. AGE contains two concrete subclasses: **`CGrammaticalCodeGenerator`** for C and **`LuaGrammaticalCodeGenerator`** for Lua. In this tutorial we will use the Lua class, defined in `LuaGrammaticalCodeGenerator.h`

A **`CodeLauncher`** takes the code generated by a **`GrammaticalCodeGenerator`**, executes it (compiling it if needed), and makes it possible to send input to the resulting program and receive its output. AGE contains two concrete subclasses: **`CCodeLauncher`** for the compiled C code and **`LuaCodeLauncher`** for the interpreted Lua code. In this tutorial we will use the Lua class, defined in `LuaCodeLauncher.h`.

The **`GrammaticalCodeFitnessEvaluator`** class has one pure abstract method (in addition to several methods inherited from the **`FitnessEvaluator`** interface), which we will have to implement:

```
private:
virtual FitnessVector  evaluateLivePopulation(
                         LivePopulation& livePopulation  ) = 0;
```

`LivePopulation` is a "running" program retrieved from a **`CodeLauncher`**, composed of *individual functions*, functions that represent the evaluated individuals. It has the following interface:

```
unsigned  size();
void      perform(void* retVals, ...);
```

The `size()` method returns the number of individuals. The `perform()` method executes the program with the specified input (supplied as variadic arguments of predefined types) and returns the output using the `retVals` pointer, which must point to a buffer for at least `size()` values of a predefined type. Each of the individual functions receives the parameters, and contributes to the output with its return value, which represents output of an individual. The types will be discussed later.

The **`evaluateLivePopulation`**`()` method must return a vector of fitness values, one for each individual in the live population.

Let's take a look at the methods the **`GrammaticalCodeFitnessEvaluator`** class implements for us:

```
protected:
void          initWithGeneratorAndLauncher(
                CodeGenerator::APtr& cg,
                CodeLauncher::APtr&  cl    );
public:
GrammaticalCodeFitnessEvaluator(
                const Grammar::Ptr& g, unsigned maxW );
virtual void  evaluate(IndividualCollection & population);
virtual bool  setsPhenotype()                        const;
virtual bool  setsDerivationInfo()                   const;
virtual const Grammar::Ptr&      grammar()           const;
unsigned                         maxWraps()          const;
virtual const IndividualDescriptor::Ptr&
                                 descriptor()        const;
```

The `virtual` methods implement the **`FitnessEvaluator`** interface. The class takes care of setting all properties of individuals including the information about the GE mapping and the phenotype, the methods **`setsPhenotype`**`()` and **`sets-DerivationInfo`**`()` consequently return `true`.

The **`evaluate`**`()` method will use a code generator and a code launcher to transform the individuals to a program represented by a **`LivePopulation`** object, which will be then passed to our **`evaluateLivePopulation`**`()` method. The code generator and code launcher to be used are passed to the initialisation method `initWithGeneratorAndLauncher()` beforehand.

The constructor takes two parameters: a pointer to the grammar, and the maximum number of wrapping events. The value of these parameters can be later retrieved using the **`grammar`**`()` and `maxWraps()` methods. The class also defines a **`descriptor`**`()` method (see **`IndividualDescriptor`** in Section 6.5.6). The descriptors return a description equivalent to the phenotype of a given individual.

Implementing the fitness evaluator

We have described the abstract base class **GrammaticalCodeFitnessEvaluator**, our evaluator will inherit from. Let's name the concrete fitness evaluator class `AntTrailFitnessEvaluator`. To complete its implementation, we need to provide a **evaluateLivePopulation**`()` method, and the methods in the **FitnessEvaluator** interface that are not implemented by **GrammaticalCodeFitnessEvaluator**:

```
virtual AntTrailFitnessEvaluator*  clone()    const;
virtual void                       init();
virtual bool                       isInited() const;
virtual FitnessOrder               order()    const;
virtual FitnessRange               range()    const;
```

and the methods in the **ArgObject** interface:

```
static FitnessEvaluator*      newObjectWithArg(
                          const std::vector<ArgValue>& argv );
static const ArgFactorySpec&  argFactorySpec();
```

We will start with the methods related to the creation of instances: **clone**`()` and **newObjectWithArg**`()`. Obviously both will need to call a constructor, the former constructs objects based on parameters of an existing objects, the latter based on user-supplied parameters. For the purposes of this tutorial, the constructor will have just one parameter, a time limit. The actual implementation in the AGE demonstration tool has additional multiple parameters.

Listing 6.6 shows and explains the constructor, the destructor and the declaration of the instance variables. Now that we have it, the implementation of the **clone**`()` method and the **ArgObject** interface is straightforward, as you can see in Listing 6.7.

So far we have implemented the model class, and a fitness evaluator that can be instantiated using the command line interface of AGE, and cloned. Now we are approaching the evaluation itself. We need to provide a glue between the model's static methods and Lua. To this purpose we define two macros of the form `AGE_LUA_WRAP_`*type*`_FUN(`*fun*`,`*luaFun*`)`. The macros expand to a definition of a function *luaFun* that pops arguments off the Lua stack, passes them to the *fun* function, and pushes the return value onto the Lua stack. One is for functions of type `int f(int)`, one is for functions of type `void f(int)`, see Listing 6.8. (The *Lua 5.1 Reference Manual* describes the convention for calling C functions in more details, search for `lua_CFunction`.)

With the aid of these two macros, we will prepare a vector containing information about these functions to be used by the **LuaCodeLauncher** as can be seen in Listing 6.9. The elements of the vector are of type **LuaFunction**, which is a simple structure defined in `LuaCodeLauncher.h`:

```
struct LuaFunction{
  typedef int (*LuaFunctionPtr) (lua_State *L);
  const char*     name;
  LuaFunctionPtr  function;
};
```

As the last step, we will implement the core methods of our fitness evaluator **`init`**`()`, **`evaluateLivePopulation`**`()`, and the related methods **`isInited`**`()`, order(), and **`range`**`()`.

In the **`init`**`()` method we will take the following preparatory steps for fitness evaluation:

(1) Set up the parameter types and return types for the individual functions: These types will be used later, when we invoke the `LivePopulation::perform()` method. In this case we want to pass a single parameter h, the trail handle of type `int`, and receive a number of pieces of food left, again of type `int`.

(2) Set up the head and tail of the individual functions to achieve the following form:

```
reset(h)
while (timeLeft(h) ~= 0) do
  (individual's phenotype)
end
return foodLeft(h)
```

(3) Create a **`LuaGrammaticalCodeGenerator`** using the above parameters, the grammar, and the maximum number of wrapping events. We could also supply a common header for all individuals: a piece of Lua code with functions that can be called from each individual. However, we do not need it in this case, as the whole model is implemented in C++.

(4) Create a **`LuaCodeLauncher`** using the vector of model functions defined earlier, and inform it about the types from step 1.

(5) Initialise our base class using the code generator and code launcher created above.

(6) Set the `inited` instance variable to `true`.

In the **`evaluateLivePopulation`**`()` method we will simply perform the individual functions of a **`LivePopulation`** and return the results (numbers of pieces of food left by each individual) as a vector of fitness values.

The implementation of these methods and the trivial methods `isInited()`, `order()`, and `range()` is shown in Listing 6.10.

We have completed the implementation of a fitness evaluator. To make it accessible from a command line tool, its class has to be added to a factory used to construct a **`Tool`** object, as shown in Listing 6.5 in Section 6.5.8.

Final notes

In this tutorial we have used the convenience classes for evaluation in Lua (**`GrammaticalCodeFitnessEvaluator`**, **`LuaGrammaticalCodeGenerator`**, and **`LuaCodeLauncher`**). Instead we could use their counterparts for C (**`CGrammaticalCodeGenerator`**, and **`CCodeLauncher`**), or we could switch between the two versions based on parameters.

Alternatively we could do without these classes, map the genotype to phenotype directly using the API described in Section 6.5.7, evaluate the phenotype in our language of choice, and directly assign the results to the individuals using the API described in Section 6.5.5.

```
#include "GrammaticalCodeFitnessEvaluator.h"          /* base class */
#include "ArgUtils.h"                                   /* ArgObject */
#include "BNFGrammar.h"
#include "yasper.h"                                  /* Grammar::Ptr */

class AntTrail;           /* forward declaration of the model class */

class AntTrailFitnessEvaluator
: public GrammaticalCodeFitnessEvaluator, public ArgObject{
  bool          inited;
  unsigned      timeLimit;
  AntTrail*     trail;
public:
  AntTrailFitnessEvaluator( unsigned  time )
  : GrammaticalCodeFitnessEvaluator(defaultGrammar(), /*maxWraps:*/3),
    inited( false ),
    timeLimit( time )
    trail( AntTrail::newAntTrail( AntTrail::santaFeMap, time ) )
  { }

  static const Grammar::Ptr&  defaultGrammar(){
    static Grammar::Ptr defaultG;
    if( ! defaultG ){
      try{
        defaultG = Grammar::Ptr( new BNFGrammar(
  "<code>      ::= <line> | <code> <line>\n"
  "<line>      ::= <condition> | <op>\n"
  "<condition> ::= if (foodAhead(h)==1) then <line> else <line> end\n"
  "<op>        ::= left(h) | right(h) | move(h)\n"                ) );
      }catch(const Cause& e){
        throw Error("AntTrailFitnessEvaluator: BNFGrammar error", e);
      }
    }
    return defaultG;
  }

  ~AntTrailFitnessEvaluator(){
    delete trail;
  }

  /* Other methods will go here. */
};
```

Listing 6.6: A skeleton of the `AntTrailFitnessEvaluator` class. The `inited` instance variable will be later used the `init()` and `isInited()` methods. To simplify the code, both the grammar and the maximum number of wrapping events are hard-coded, and the methods defined inline. Note that the grammar includes an `h` parameter for each of the ant's actions, we will need that to pass the trail handle to the static methods of the `AntTrail` class. The time limit is stored in an instance variable for later use by the `clone()` method.

```
AntTrailFitnessEvaluator*  clone()  const{
  return new AntTrailFitnessEvaluator( timeLimit );
}

const ArgFactorySpec&      argFactorySpec(){
  static ArgFactorySpec spec = {
                         "ant", /*mandatoryArgCount:*/ 0,
                         ArgSpecVector()                 };
  if( spec.argSpecs.empty() ){
    ArgSpec  as;
    as.label           = "time(600)";
    as.type            = UIntArg;
    as.uintRange.from  = 0;
    as.uintRange.upto  = UINT_MAX;
    spec.argSpecs.push_back(as);
  }
  return spec;
}

FitnessEvaluator* newObjectWithArg(const std::vector<ArgValue>& argv){
  if(argv.size() == 0)
    return new AntTrailFitnessEvaluator( /*default time limit:*/ 600);

  assert(argv.size() == 1);
  return new AntTrailFitnessEvaluator( argv.at(0).uintValue );
}
```

Listing 6.7: Implementation of the **`ArgObject`** and **`Cloneable`** interfaces in the `AntTrailFitnessEvaluator` class. A single parameter can be supplied from the command line, a time limit with a default value of 600.

```
#include "lua.hpp"                 /* Lua interface for C++ */

#define AGE_LUA_WRAP_INT_TO_INT_FUN(FUN, LFUN)               \
  static int LFUN( lua_State* lua);                          \
  static int LFUN( lua_State* lua){                          \
    lua_pushinteger(  lua, FUN( lua_tointeger(lua, -1) )  ); \
    return 1;                                                \
  }

#define AGE_LUA_WRAP_INT_TO_VOID_FUN(FUN, LFUN) \
  static int LFUN( lua_State* lua);            \
  static int LFUN( lua_State* lua){            \
    FUN( lua_tointeger(lua, -1) );             \
    return 1;                                  \
  }
```

Listing 6.8: Macros for making C functions accessible from Lua.

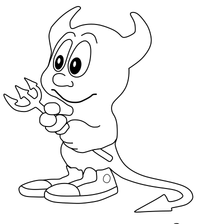
```
#include "App-AntTrail.h"        /* AntTrail, the model class */

AGE_LUA_WRAP_INT_TO_VOID_FUN( AntTrail::left,      l_left      )
AGE_LUA_WRAP_INT_TO_VOID_FUN( AntTrail::right,     l_right     )
AGE_LUA_WRAP_INT_TO_VOID_FUN( AntTrail::move,      l_move      )
AGE_LUA_WRAP_INT_TO_INT_FUN(  AntTrail::foodAhead, l_foodAhead )
AGE_LUA_WRAP_INT_TO_INT_FUN(  AntTrail::foodLeft,  l_foodLeft  )
AGE_LUA_WRAP_INT_TO_INT_FUN(  AntTrail::foodEaten, l_foodEaten )
AGE_LUA_WRAP_INT_TO_INT_FUN(  AntTrail::timeLeft,  l_timeLeft  )
AGE_LUA_WRAP_INT_TO_VOID_FUN( AntTrail::reset,     l_reset     )

const LuaFunction  AntTrailFitnessEvaluator :: antTrailFunctions[] = {
  {"left", &l_left},          {"right", &l_right},
  {"move", &l_move},          {"foodAhead", &l_foodAhead},
  {"foodLeft", &l_foodLeft},  {"foodEaten", &l_foodEaten},
  {"timeLeft", &l_timeLeft},  {"reset", &l_reset}
};

const std::vector<LuaFunction>
  AntTrailFitnessEvaluator :: antTrailFunctionVector(
  antTrailFunctions,  antTrailFunctions +
  sizeof(antTrailFunctions)/sizeof(antTrailFunctions[0])
);
```

Listing 6.9: Wrapping the static methods and putting them in a vector together with the names under which they will be accessible from Lua.

```
#include "LuaGrammaticalCodeGenerator.h"
#include "LuaCodeLauncher.h"

void             init(){
  CodeGenerator::APtr  cg;
  CodeLauncher::APtr   cl;
  const CodeType                  returnType(   CodeTypeInt );
  const std::vector<CodeType>     paramTypes(1, CodeTypeInt );
  const std::vector<const char*>  paramNames(1, "h" );

  cg = CodeGenerator::APtr( new LuaGrammaticalCodeGenerator(
                                  grammar(),
                                  maxWraps(),
                                  /*common header:*/ "",
                                  paramNames,
                                  /*head:*/
                                    "\treset(h)\n"
                                    "\twhile (timeLeft(h) ~= 0) do\n",
                                  /*tail:*/
                                    "\n"
                                    "\tend\n"
                                    "\treturn foodLeft(h)"        ) );
  cl = CodeLauncher::APtr( new LuaCodeLauncher(
                                 LuaLivePopulation::UseUnsafeMath,
                                 antTrailFunctionVector       ) );

  cl->setIndividualTypes( returnType, paramTypes );
  initWithGeneratorAndLauncher( cg, cl );
  inited = true;
}

bool           isInited()    const{
  return inited;
}
FitnessOrder   order()       const{
  return LowerBetter;
}
FitnessRange   range()       const{
  return FitnessNonnegative;
}

FitnessVector  evaluateLivePopulation(LivePopulation& livePopulation){
  std::vector<int>  foodLeft( livePopulation.size() );

  livePopulation.perform( & foodLeft.front(), trail->handle() );

  return FitnessVectorFromInts( FitnessNonnegative, foodLeft );
}
```

Listing 6.10: Implementation of the `AntTrailFitnessEvaluator` core using **`LuaGrammaticalCodeGenerator`** and **`LuaCodeLauncher`**.

6.7 Licence


BSD Daemon[2]

AGE is opensource and can be used for personal, academic, and commercial purposes at no cost.

AGE itself is licensed under the terms of the standard three-clause BSD licence. Its full text follows:

The text of this thesis is licensed under the *Creative Commons Attribution–Noncommercial–No Derivative Works 3.0 Czech Republic Licence*, see page 5.

This distribution also includes other software distributed under the BSD licence or similarly liberal licences, namely:

Lua: A lightweight scripting language.
Copyright © 1994–2008 Lua.org, PUC-Rio.
MIT licence.
Full copyright and licence in `src/lua-5.1.4/COPYRIGHT`.

[2]BSD Daemon Copyright © 1988, by Marshall Kirk McKusick. All Rights Reserved. The BSD Daemon is the mascot of the Berkeley Software Distribution, a major opensource Unix distribution. AGE uses tidbits of opensource code from modern BSD descendants for portability and itself has the BSD licence, commonly used for opensource software developed in the academic environment.

getopt(): Standard command option parsing.
Copyright © 1987, 1993, 1994 The Regents of the University of California.
From Apple Darwin Libc.
BSD licence (three-clause).
Full copyright and licence in `src/getopt.c`.

popen(): A bidirectional, thread-safe implementation of `popen()`.
Copyright © 1988, 1993 The Regents of the University of California.
Based on code from NetBDS. Modified by Adam Nohejl.
BSD licence (three-clause).
Full copyright and licence in `src/PopenFix.c`.

pstdint.h: A portable `stdint.h`.
Copyright © 2005–2007 Paul Hsieh.
BSD licence (three-clause).
Full copyright and licence in `src/pstdint.h`.

random_r: An improved random number generation package.
Copyright © 1983 Regents of the University of California.
Based on code from uClibc. Modified by Adam Nohejl.
BSD licence (three-clause).
Full copyright and licence in `src/Random.c`.

yasper (a modified version of): A non-intrusive reference counted pointer.
Copyright © 2005–2007 Alex Rubinsteyn.
Modified by Adam Nohejl.
zlib/libpng licence.
Full copyright and licence in `src/yasper.h`.

Chapter 7

Developer Documentation

Because AGE is a framework, not a closed tool, its developer documentation overlaps with its user documentation. In fact, we have documented its design in Chapter 5, and its principal classes and the interactions between them in Chapter 6. I believe that what is left is best documented by source code comments.

In this chapter we will cover two places, where the implementation needs more than a few lines of a commentary. The following notes will be useful for those who would either like to modify the source code, or verify that it works correctly.

7.1 Selection schemes

Tournament selection (described in Section 2.6.3) is possibly one of the few algorithm elements we have mentioned whose implementation does not follow directly from its definition. If we pick the competitors for the tournament without replacement, we are essentially shuffling, or permuting, the population. There is a very simple and correct way to do this. The algorithm, called sometimes the Fischer-Yeats shuffle, is described and explained by Cormen et al. (2001, p. 103). I reproduce their pseudocode for shuffling an array A:

```
RANDOMIZE-IN-PLACE($A$)
1  $n \leftarrow length[A]$
2  for $i \leftarrow 1$ to $n$
3        do swap $A[i] \leftrightarrow A[\text{RANDOM}(i, n)]$        $\triangleright$ $a \leq \text{RANDOM}(a, b) \leq b$
```

As can been seen, after iteration i the element $A[i]$ is never altered. This fact can be used for picking only a certain number n of random elements: tournament competitors. It is also useful to realise that the algorithm is correct regardless of the initial order of the array A: we do not need to restore the order for each tournament selection.

Some other seemingly flawless methods for shuffling are not correct (such as PERMUTE-WITH-ALL constructed as a non-working example by Cormen et al., 2001, exercise 5.3-3, p. 105). One such method is used for tournament selection in the SGA-C environment (see footnote 3 on page 33). AGE implements tournament selection using the correct shuffling algorithm taking advantage of the two above mentioned facts.

Roulette-wheel selection, on the other hand, is very simple to implement correctly. One should, however, realise that it can be implemented using binary search reducing it from $O(n)$ to $O(\log n)$ time. This is the way it is implemented in AGE.

7.2 Fast bit-level mutation

In Section 5.4, we have argued that bit-level mutation could be implemented efficiently using a statistic approximation, very likely eliminating the need for codon-level mutation. We will explain how exactly fast bit-level mutation is implemented in AGE, and compare its performance to the naïve, slow bit-level implementation and codon-level mutation.

The decision whether to mutate a single bit follows a Bernoulli distribution with parameter p_m (the probability of mutation), the number of unaltered bits until the next mutation follows a geometric distribution with the same parameter, symbolically:

$$\begin{aligned} X &\sim \mathrm{Geom}(p_m) \\ f(k) &= P[X = k] = (1-p)^k p \\ F(k) &= \sum_{i=0}^{k} f(i) = \sum_{i=0}^{k} (1-p)^i p = 1 - (1-p)^{k+1} \end{aligned}$$

where f is the probability mass function, and F is the cumulative distribution function, both defined as discrete, which is sufficient for our needs. Random variates from this distribution can be and efficiently generated using the inversion method. The method is described in more details by Devroye (1986, ch. 2, ch. 3) along with a proof of its correctness. The gist of it is that if we appropriately define F^{-1}, an inverse of F, then $F^{-1}(U)$ follows the distribution defined by F, provided that the random variable U is uniformly distributed over $[0, 1]$. This can be directly used to generate a random variate with distribution F if there is a computational formula for F^{-1}. The inverse of a distribution F for the geometric distribution $\mathrm{Geom}(p_m)$ can be computed easily:

$$F^{-1}(u) = \frac{\log(1-u)}{\log(1-p_m)}$$

To generate a random variate k of a geometric distribution with parameter p_m, which is from $\{0, 1, 2, \ldots\}$, we use an adjusted formula:

$$k = \left\lfloor \frac{\log(1-u)}{\log(1-p_m)} \right\rfloor$$

where u is a random variate from the uniform distribution over $[0, 1)$.[1] This formula is defined, and correct, for $u \in [0, 1)$, and $p_m \in (0, 1)$. The special cases $p_m \in \{0, 1\}$ do not actually require random variate generation.

[1]The formula I use is slightly different from the one presented by Devroye (1986): $\lceil \log(1-u)/\log(1-p) \rceil$, further simplified to $\lceil \log(u)/\log(1-p) \rceil$ under the assumption that $0 < u < 1$, uniformly distributed. The difference, however, is only technical because (1) I use $u \in [0, 1)$, as such an uniform random variate can be more conveniently generated, (2) I define $X \sim \mathrm{Geom}(p)$ as having values from $\{0, 1, 2, \ldots\}$, not $\{1, 2, 3, \ldots\}$.

This value k, the number of bits until the next mutated bit, can be generated in constant time, by calling the random number generator once to generate u and then applying the above formula. The computational complexity of applying the mutation to a string of n bits with probability p_m is then in average $O(np_m)$, while using the naïve bit-by-bit implementation it is $O(n)$. Obviously, the complexity of codon-level mutation is $O(n/c)$, where c is the codon size. All three mutations are asymptotically equal, but the fast bit-level mutation will be faster by a constant factor for a sufficiently low probability of mutation p_m.

It is also worth noting that the geometric distribution is *memoryless*, which means that it does not matter whether we draw a new value k and start counting over again at each chromosome boundary, or whether we apply the mutation as if the whole population was concatenated into a single giant chromosome. In both cases the distribution of the number of bits between each two consecutive mutations will be geometric. In AGE, the one-giant-chromosome approach is used. It is slightly more difficult to implement, but can reduce the number of RNG calls significantly for low mutation rates and short chromosomes.

8-bit codons:

p_m	naïve bit-level	fast bit-level	codon-level
1 %	1.50 s (1×)	0.09 s (0.06×)	0.18 s (0.12×)
2 %	1.51 s (1×)	0.19 s (0.12×)	0.18 s (0.12×)
5 %	1.56 s (1×)	0.46 s (0.29×)	0.19 s (0.12×)

31-bit codons:

p_m	naïve bit-level	fast bit-level	codon-level
1 %	5.56 s (1×)	0.39 s (0.07×)	0.17 s (0.03×)
2 %	5.62 s (1×)	0.79 s (0.14×)	0.17 s (0.03×)
5 %	5.83 s (1×)	1.96 s (0.33×)	0.18 s (0.03×)

Table 7.1: Comparison of execution times of three mutation operators in AGE. The operators are applied 1000 times to a population of 100 randomly initialised chromosomes of lengths evenly distributed between 100 and 200 codons, inclusive, in the same environment as used for the experiments in Chapter 8.

Table 7.1 compares performance of the three versions of mutation as implemented in AGE, when applied with probabilities of 1 to 5 % on 8-bit and 31-bit codons. For the more usual 8-bit codons, the fast bit-level mutation is always significantly faster than the naïve version, at 1 % it is faster than the codon-level mutation, at 2 % it has virtually the same running time. For the 31-bit codons, the codon-level mutation is always the fastest one.

The fast bit-level mutation provides a performance gain of an order of magnitude for low mutation rates (1 to 2 %) and the standard codon size (8 bits), which will in most cases make its running time negligible relatively to the overall running time of the evolutionary algorithm. Further gains from the codon-level mutations at higher rates and codon sizes are therefore usually inconsequential. It would be possible to implement a fast codon-level mutation in a similar manner, which would, for the same reason, only make sense for very high mutation rates.

This, of course, does not tell us anything about the effect of of codon-level mutation compared with bit-level mutation. In Section 5.4 we have argued that there is no apparent benefit in using codon-level mutation, in Chapter 8 we will back that up by experiments in two different applications.

Chapter 8

Experiments

I will examine several benchmark applications of grammatical evolution, either previously used in the literature (O'Neill and Ryan, 2003) or in other software (O'Neill et al., 2008; Nicolau and Slattery, 2006). In each application I will try to replicate the results of others using a similar setup, and compare the results. Based on the statistics from the initial experiment, several adjustments to the parameters and the algorithm itself will be explored.

The goal is to show that AGE is able to produce results, not necessarily better than, but comparable with those of other implementations, and to demonstrate how it helps analyse the results and thus guide further experimentation.

8.1 Methodology

Summary of each experiment is presented in a table derived from the "tableau" format used by Koza (1992), and in a modified form by O'Neill and Ryan (2003). In addition to the entries used by either of them, I also specify the algorithm (entries *Algorithm*, *Initialisation*, *Selection*, *Operators*, *GE Mapping*).

I always compare data from multiple runs, either 200 or 1000. (O'Neill and Ryan (2003) use always 100 runs, Koza (1992) used hundreds of runs.) In case of AGE these are always runs from one sequence with RNG seed 42, in case of GEVA these are runs with RNG seeds $1, 2, \ldots, n$.

Whenever I describe a difference of two sets of numerical results (or rates of success in two sets) as *significant*, it means that the statistical hypothesis that the two sets are samples of a statistical population with the same mean (or that the probabilities of success in the two sets are the same), has to be rejected on the level of statistical significance of 5 %. Accordingly, whenever a difference is said to be *insignificant*, the same hypothesis has to be rejected on the level of 5 %.

I use Student's t-test for testing equivalence of means, and a simple test of equal proportions for testing the probabilities of success.[1]

As the most important measure of success I adopt the cumulative rate of finding the solution over a number of runs, as used in the experiments by O'Neill and Ryan (2003). The solution is assumed to be found when the fitness of the

[1]Both are computed using the *R stats package* (functions `t.test` and `prop.test`), which is part of the open-source *R Project for Statistical Computing* available at `http://www.r-project.org/`.

best individual of a given generation reaches a certain level. The level is chosen separately for each application so that is ensures the solution has really been found. For the purpose of these test, however, I let the algorithms continue to run even after finding the solution.

As a measure of performance I use the CPU time consumed by a number of runs of an algorithm as reported by the standard UNIX `time` utility (user and system time combined). If multiple invocations of the program are needed, their times measured in hundredths of seconds are added up. Note that the "wall clock" running time compared to the CPU time may be either lower, when multiple CPUs or cores are used, or higher, when other the CPU is shared with other processes.

All times have been measured on a 2.4 GHz Intel Core 2 Duo machine with 2 GB of RAM (Apple MacBook MB467LL/A), on Mac OS X 10.5.7. AGE has been compiled using Apple's e version 4.0.1 with optimisation flag `-O3`,[2] GEVA has been compiled and executed on Java SE 1.6.0_07 HotSpot 64-Bit Server VM with flags `-Xms256m -Xmx1024m`.[3] Assertions are disabled (`-DNDEBUG` for C++, default for Java). Optimised settings for both the C++ compiler and the Java Virtual Machine are used in order to give equal conditions to both implementations.

8.2 Symbolic regression

Symbolic regression has been already discussed as an example problem in Section 2.5. In this experiment we will use the same target function as in the symbolic regression example supplied with GEVA (O'Neill et al., 2008, sec. 6.6). Initially we will also use a very similar setup and try to replicate the results achieved by GEVA.

8.2.1 Initial experiment

We first need to examine the algorithms and methods used in GEVA, which we already did in Section 3.2. GEVA's configuration file for the symbolic regression example (`SymbolicRegression.properties`) tells us which parameters and algorithm elements are used, including the parameters of the GE mapping: the grammar and the maximum number of wrap events. We have already said that GEVA uses a remarkably large codon size of 32 bits and suggested that while it may suit well the codon-level mutation operator, it is an unnecessarily high

[2]This GCC optimisation level enables inlining, and therefore provides the best result for AGE's C++ code, which is based on the Standard Template Library. The only exception to this setting is the Lua library used by AGE, which is compiled separately with its own default optimisation flag `-O2`.

[3]The settings were chosen according to the *Java Tuning White Paper* available at `http://java.sun.com/performance/reference/whitepapers/tuning.html`. The machine used for testing is considered "server-class" by the Java Virtual Machine (JVM), therefore the flags `-server` and `-XX:+UseParallelGC` are enabled implicitly. Other optimisation options suggested by the white paper, namely `-Xmn`*n*, `-XX:+UseBiasedLocking`, `-XX:+AggressiveOpts` were tested, but did not have any noticeable effect on GEVA using this release of JVM. Version 1.6.0_07 is the latest release available for Mac OS X as of this writing.

value. While codon size cannot be changed in GEVA, we can compare results of different codon sizes using AGE.

Second, we need to examine the fitness function used in the example supplied with GEVA:

- The fitness function uses JScheme to evaluate expressions, and therefore the grammar used for symbolic regression is constructed to generate prefix expressions (see Listing 8.1). We construct an equivalent grammar for infix expressions suitable for Lua or C to be used in AGE (see Listing 8.2). This is merely a technical difference, which can affect performance, but not the results.
- The fitness cases are chosen randomly from the interval $[-1, 1)$ each time an individual is evaluated. This is a highly unusual approach. Koza (1992, sec. 6.4.1) suggests that fitness cases might vary between generations, as opposed to between individuals, but remarks that it prevents reuse of previously computed values. When fitness cases vary within one generation, it is questionable whether the computed values can be compared with each other, and thus used effectively for selection. Additionally, GEVA caches fitness values based on phenotype (see Section 3.2, this caching is enabled for the symbolic regression example), which entirely defeats any purpose the varying fitness cases might have, as the changed fitness cases are never applied to individuals who survived from the previous individuals. Fitness is the sum of squared errors over the fitness cases.

As AGE does not use any caching mechanism, and it is unclear how the random fitness cases affect performance, GEVA will be tested (1) with its initial setup, (2) without caching, and (3) with fixed fitness cases. Similarly AGE will be tested with both (1) 31-bit codons (of the supported sizes the one closest to 32 bits in GEVA), and (2) the usual 8-bit codons :

– *GEVA (1):*	codon size: 32, random fitness cases, cached.
– *GEVA (2):*	codon size: 32, random fitness cases, not cached.
– *GEVA (3):*	codon size: 32, fixed fitness cases, cached.
– *AGE (1):*	codon size: 31, fixed fitness cases.
– *AGE (2):*	codon size: 8, fixed fitness cases.

The remaining parameters are summarised in Table 8.1:

```
<expr>  ::= ( <op> <expr> <expr> ) | <var>
<op>    ::= +|-|*
<var>   ::= x0|1.0
```

Listing 8.1: Simple grammar for symbolic regression in JScheme (prefix expressions), used in GEVA.

Objective:	Find a function of one variable x represented by an expression to fit a given set of the target function values at specified points. The target function is $x^4 + x^3 + x^2 + x$.
Terminal operands:	x, 1.0.
Terminal operators:	$+$, $-$, $\cdot$ (all binary).
Fitness cases:	*GEVA (1–2):* 20 random points from the interval $[-1, 1)$. *GEVA (3), AGE (1–2)*: 20 fixed points (-1.0, -1.1, ..., 1.9).
Raw fitness:	The sum of squared errors taken over the 20 fitness cases.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple with elite size: 10, generations: 101, population: 500.
Initialisation:	Random, fixed chromosome length: 200.
Selection:	Tournament, size: 3.
Operators:	Codon-level mutation, probability: 0.02. Fixed-length one-point crossover, probability: 0.9.
GE mapping:	Grammar: see Listing 8.1 and 8.2. Maximum wraps: 3. Codon size: 32, 31, 8 for *GEVA (1–3), AGE (1), AGE (2)* respectively.
Success predicate:	Raw fitness lower than 0.00001. (Effectively ensures equivalence with the target function regardless of the fitness cases.)

Table 8.1: Symbolic regression, parameters for tests *GEVA (1–3)*, *AGE (1–2)*.

```
<expr>  ::= ( <expr> <op> <expr> ) | <var>
<op>    ::= + | - | *
<var>   ::= x | 1.0
```

Listing 8.2: Simple grammar for symbolic regression in Lua or C (infix expressions), used in AGE, equivalent to the grammar in Listing 8.1.

8.2.2 Comparison of results with GEVA

As can be seen in Figure 8.1, the original setup of GEVA fared the worst. The second setup, without caching, showed only a mild improvement, having a significantly higher success from generation 33 onward. This, however, was offset by an almost doubled running time. The third setup, with a success rate significantly higher than the original from generation 6 onward, achieved the highest success rate for GEVA without any remarkable performance penalty. Success rates in the last generations of these two setups were 249, 255, and 404 for *GEVA (1)*, *(2)*, and *(3)*. As we can see the use of fitness cases randomly varied within in a population reduces the success rate drastically, with or without caching.

The two setups of AGE have both achieved significantly higher success rates than any setup of GEVA in all generations, eventually reaching rates of 706 and 701. The results achieved with 8-bit codons are significantly better in the generations 4 to 89, and significantly worse only in the last four generations. While the performance was not affected by the use of 31-bit codons, there is no evidence that the larger codons are of any substantial benefit in this application.

Given the same algorithms and parameters (except the codon sizes 31 and 32), the large difference in success rate between GEVA and AGE is curious. It

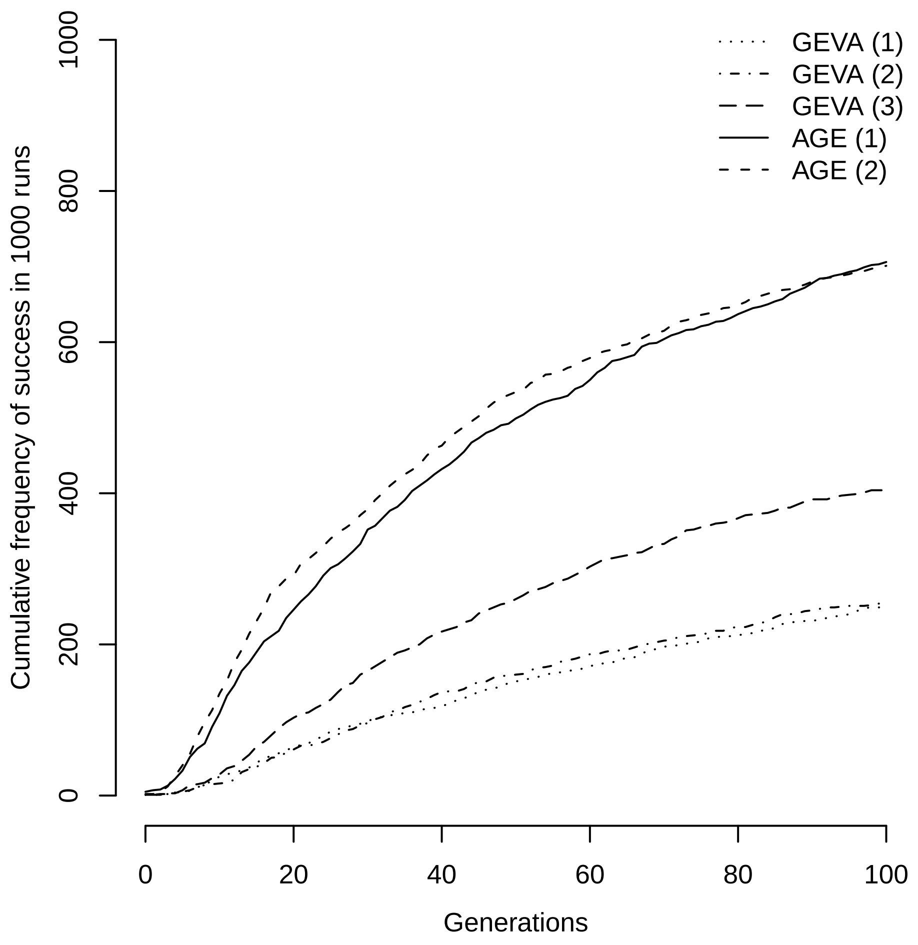


Figure 8.1: Cumulative frequency of success of AGE and GEVA in symbolic regression.

suggests that there are important differences in implementation, not obvious from the documentation of GEVA.

The statistics from generation 0 (Table 8.2), before applying any operators and selection, show that both GEVA and AGE start with roughly the same population characteristics as expected. This further supports that the algorithms behave differently despite the same settings.

The initial tests have proved that AGE produces results competitive with GEVA. The performance is compared in Figure 8.2. Both setups of AGE are approximately 10 times faster than the original setup of GEVA.

	GEVA (3)	*AGE (1)*	*AGE (2)*
Invalid individuals	7.94	7.41	7.41
Best fitness	7.20	5.33	5.21
Worst fitness	†	2606.61	1763.05
Average fitness	60.92	63.57	54.69
Variance of fitness	$6.49 \cdot 10^5$	$5.38 \cdot 10^6$	$3.69 \cdot 10^5$
Average chromosome length	200	200	200
Average used codons	15.20	15.29	15.29
Average wrap events	†	0	0
Average tree height	4.09	4.12	4.12

† Data not available.

Table 8.2: Symbolic regression, comparison of statistics from generation 0 (averaged over 1000 runs).

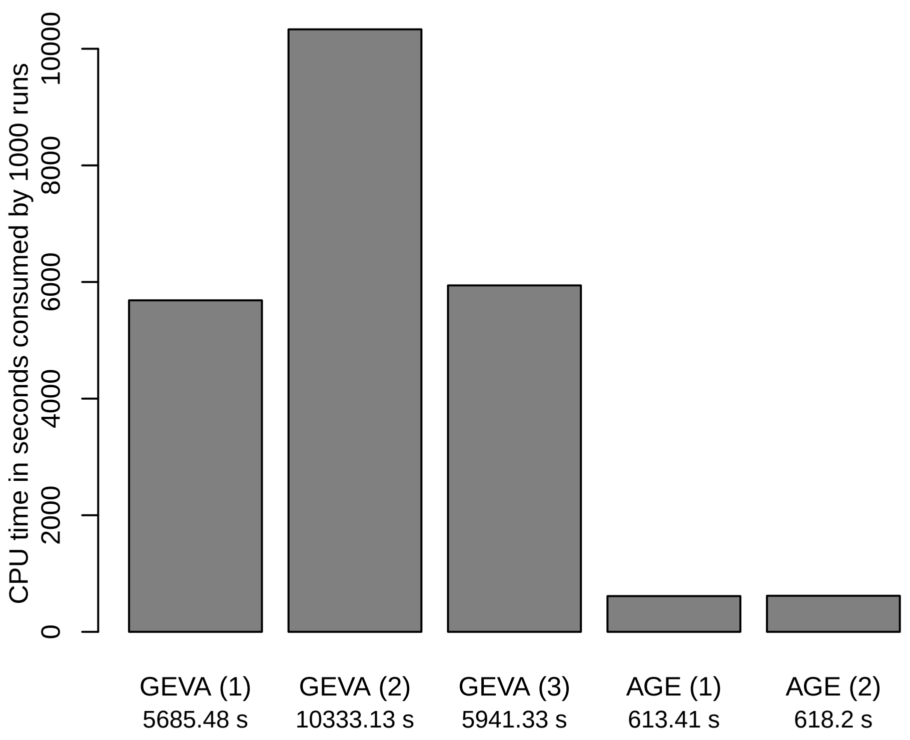


Figure 8.2: Running times of AGE and GEVA for symbolic regression.

8.2.3 Further experiments

In the following set of experiments we will explore several adjustments to the initial setup, which mimicked GEVA. We will be interested in effects of the following changes:

(1) roulette-wheel selection with a scaling instead of tournament selection,
(2) variable-length instead of fixed-length chromosomes,
(3) bit-level instead of codon-level mutation,

We will use the initial setup *AGE (2)* as our starting point, and in each of the following configurations we will modify one more parameter or algorithm element:

- *AGE (1′):* roulette-wheel selection, reversal and linear scaling.
- *AGE (2′):* variable-length crossover, random initialisation to 100–150 codons.
- *AGE (3′):* bit-level mutation with probability of 1.5 %.

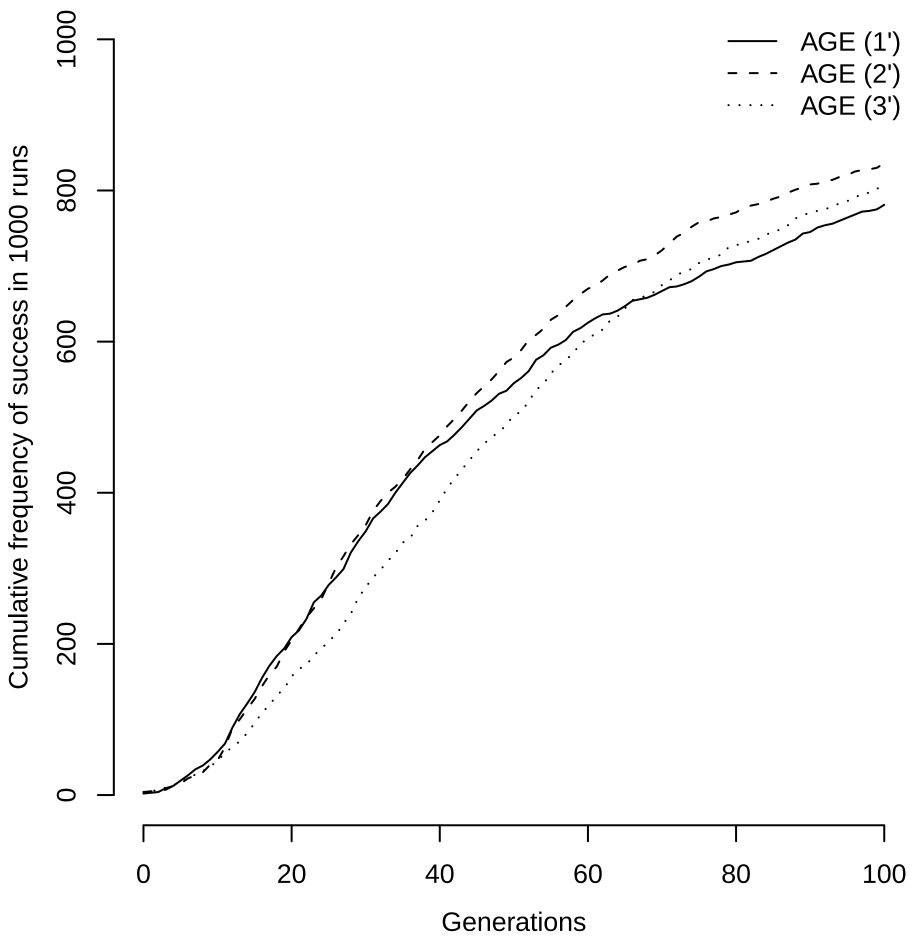


Figure 8.3: Cumulative frequency of success of AGE (alternative parameters) in symbolic regression.

As we can see in Figure 8.3, all three setups show a significant improvement, thanks to the change of selection scheme and addition of a scaling. I have not been able to reach similar results using tournaments of any size (with other parameters fixed).

The variable-length chromosomes give us an additional significant improvement, albeit a smaller one.

The bit-level mutation resulted has a small but significant adverse effect. The problem is that, as we have said earlier (Section 5.4), the rates of bit-level and codon-level mutation do not correspond to each other. If we have chosen exactly the same rate as for codon-level mutation the effective mutation rate would be too high in this case. We have tried a lower one, which had a similar but still different outcome. But what is more important, the fast bit-level mutation had a 16 % lower running time. The times were 577.15 s for *AGE (1′)*, 628.77 s for *AGE (2′)*, and 533.98 s for *AGE (3′)*. The setups *AGE (1′)* and *AGE (3′)* were also faster than the initial experiment (618.2 s), while providing better results.

8.3 Santa Fe ant trail

The Santa Fe ant trail is another benchmark application of evolutionary algorithms. Koza (1992) used it for experiments with tree-based GP citing previous application of genetic algorithms to the problem. Both GEVA and libGE are accompanied by an implementation of this application.

The goal is to navigate an artificial ant so that it finds all food lying on a rectangular grid within a time limit. The ant's repertoire of actions is limited:

- Left turns the ant left by 90 ° without moving,
- Right turns the ant right by 90 ° without moving,
- Move advances the ant one square forward in the direction it is facing, eating any food on that square,
- Food-Ahead tests the square the ant is facing for food.

All actions, except the last one, take one unit of time. The grid is wrapped around a toroidal plane to save the ant from falling off its edge. At the beginning the ant is placed in the north-west corner facing its first piece of food located on the east. Other pieces of food are placed to form a trail. The Santa Fe ant trail, shown in Figure 8.4, is a particular placement of 89 pieces of food on a 32 by 32 grid, which contains an increasingly demanding sequence of gaps and turns.

In genetic programming, the solution has a form of a short program,[4] which is executed in a loop until time runs out. The program is composed of the following statements:

- actions Left, Right, or Move,
- conditional statements of in the form **if** foodAhead **then** $block_1$ **else** $block_2$,
- blocks formed from the previous types of statements.

Koza (1992) built blocks of two or three statements in his tree-based GP. O'Neill and Ryan (2003) use a grammar (Listing 8.3) that can build blocks by adding one action or a conditional statement at a time, but does not allow blocks inside conditional statements, reducing them to

[4]Koza (1992) mentions other, previously used, forms of a solution: a finite automaton represented as a fixed-length binary string evolved using GA, and a neural network.

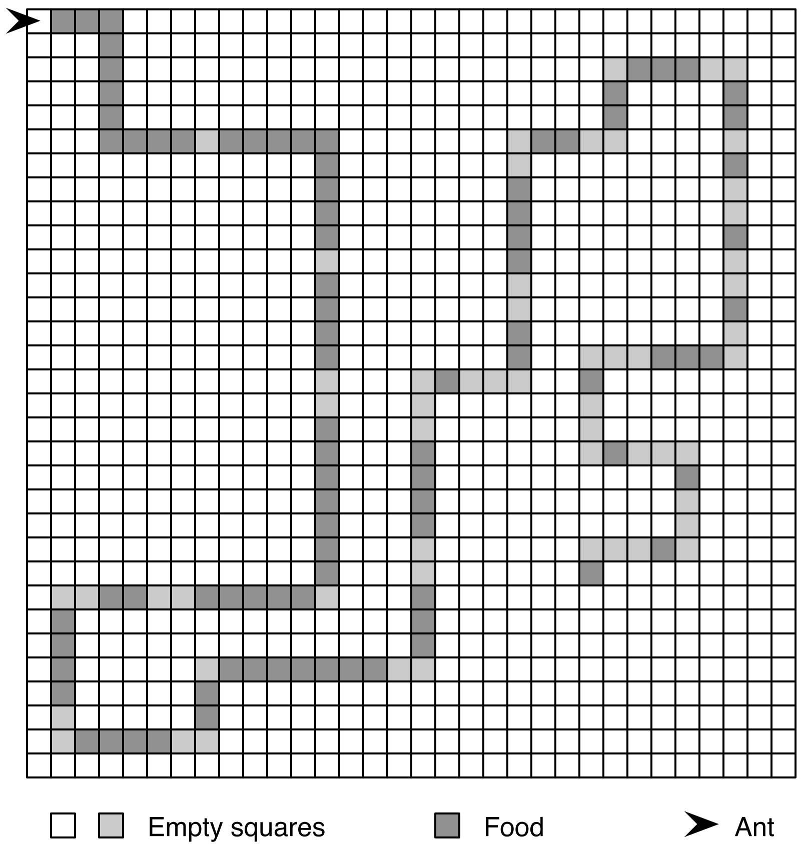


Figure 8.4: The Santa Fe ant trail. 89 pieces of ant food on a 32 by 32 toroidal grid.

if FOODAHEAD **then** *action-or-conditional*$_1$ **else** *action-or-conditional*$_2$.

The same grammar is used in an example that accompanies libGE. The Santa Fe ant trail application supplied with GEVA uses yet another grammar (Listing 8.4), which allows only sequences of actions in conditional statements:

if FOODAHEAD **then** *list-of-actions*$_1$ **else** *list-of-actions*$_2$.

```
<code>           ::= <line> | <code> <line>
<line>           ::= <if-statement> | <op>
<if-statement>   ::= if (foodAhead(h)==1) then <line> else <line> end
<op>             ::= left(h) | right(h) | move(h)
```

Listing 8.3: A grammar for the ant trail problem. A Lua equivalent to the C-like grammar used by O'Neill and Ryan (2003) and in a libGE example.

We will only use the grammar used in GEVA in our experiments, but it is interesting to note that while the choice of representation in the traditional GP is limited, grammatical evolution gives virtually endless possibilities. Some grammars may only result in a preference for certain constructions, some like these two examples may even preclude some constructions from being used. Additionally, the structure of the grammar affects the mapping process.

We will initially attempt to replicate the results achieved with GEVA.

```
<prog>       ::= <code>
<code>       ::= <line> | <code> <line>
<line>       ::= <condition> | <op>
<condition>  ::= if (foodAhead(h)==1) then <opcode> else <opcode> end
<op>         ::= left(h) | right(h) | move(h)
<opcode>     ::= <op> | <opcode> <op>
```

Listing 8.4: Another grammar for the ant trail problem. A Lua equivalent to the Groovy grammar used in a GEVA example.

8.3.1 Initial experiment

The initial setup is based on the parameters from GEVA's configuration file for the Santa Fe ant trail example (`SantaFeAntTrail.properties`). This configuration involves the "sensible" initialisation method. We will also perform the same tests with the random initialisation method, for reasons that will be explained after we analyse the results of "sensible" initialisation in GEVA.

The fitness function used in GEVA employs the Groovy scripting language to evaluate expressions. As in the previous example, we will use an equivalent grammar for Lua in AGE (see Listing 8.4). The fitness consists, as usual for the ant trail problem, of a single fitness case, and its value is equivalent to the number of pieces of food left after the limit of 600 time units. The same fitness function will be used in AGE. In this case the fitness value can be naturally formulated as the amount of food either left or consumed. The difference is only technical.

As before, AGE will be tested with both (1) 31-bit codons (of the supported sizes the one closest to 32 bits in GEVA), and (2) the usual 8-bit codons, resulting in the following series of tests:

- *GEVA (s):* codon size: 32, "sensible" initialisation.
- *GEVA (r):* codon size: 32, random initialisation with length 20.
- *AGE (s-1):* codon size: 31, "sensible" initialisation.
- *AGE (s-2):* codon size: 8, "sensible" initialisation.
- *AGE (r-1):* codon size: 31, random initialisation with length 20.
- *AGE (r-2):* codon size: 8, random initialisation with length 20.

The remaining parameters, which were based on the configuration of GEVA, are summarised in Table 8.3,

8.3.2 Comparison of results with GEVA

In this experiment, we will compare results from 200 runs due to the long running time of GEVA. I initially tested only the sensible initialisation setups *GEVA (s)* and *AGE (s-1–2)*. Neither GEVA, nor AGE with 31-bit and 8-bit codons achieved impressive results, succeeding 5 times, twice, and once in 200 runs. The results of GEVA were significantly better than those of AGE with 31-bit and 8-bit codons from generation 39 and 41, but as we will see, GEVA did not use the specified initialisation method, invalidating this comparison.

To discover what might have caused the difference between AGE and GEVA, I have, as in the previous experiment, examined the statistics from both tools. The statistics from generation 0 (Table 8.4) have shown exactly the same numbers from the two sets of runs of AGE. This has been expected as a result of

Objective:	Find a program for navigating the ant so that it finds all 89 pieces of food on the Santa Fe trail within 600 time units.
Terminal operands:	None.
Terminal operators:	LEFT, RIGHT, MOVE, FOOD-AHEAD.
Fitness cases:	One fitness case.
Raw fitness:	Number of pieces of food left on the grid after 600 time units of running the ant's program in a loop.
Scaled fitness:	Same as raw fitness.
Algorithm:	Simple with elite size: 10, generations: 101, population: 100.
Initialisation:	Ramped ("sensible") with maximum height 6 for *GEVA (s), AGE (s-1–2)*. Random with length 20 for *GEVA (r), AGE (r-1–2)*
Selection:	Tournament, size: 3.
Operators:	Codon-level mutation, probability: 0.01. Fixed-length one-point crossover, probability: 0.9.
GE mapping:	Grammar: see Listing 8.4, Maximum wraps: 3. Codon size: 32, 31, 8 for *GEVA (1–2), AGE (s-1, r-1), AGE (s-2, r-2)* respectively.
Success predicate:	Raw fitness equivalent to 0. (All food eaten.)

Table 8.3: Santa Fe ant trail, parameters for tests *GEVA (1–2)*, *AGE (s1–2)*, *AGE (r1–2)*.

the "sensible" initialisation method. Additionally, each production rule used is represented by one codon: all codons should be used exactly once when mapping the initial population.[5] In the statistics from AGE, this is correctly reflected by exactly the same value of the average chromosome length and the average number of used codons 4.97.

	GEVA (s)	*AGE (s-1)*	*AGE (s-2)*
Invalid individuals	0	0	0
Best fitness	71.68	76.07	76.07
Worst fitness	†	89.00	89.00
Average fitness	88.11	88.47	88.47
Variance of fitness	7.89	4.91	4.91
Average chromosome length	(!) 8.52	4.97	4.97
Average used codons	(!) 9.01	4.97	4.97
Average wrap events	†	0	0
Average tree height	5.65	4.72	4.72

† Data not available.

Table 8.4: Santa Fe ant trail, comparison of statistics from generation 0 (averaged over 200 runs).

GEVA, however, reports a higher number of used codons (9.01) than chromosome length (8.52). This is possible only if at least some of the individuals

[5]In libGE, there is an option to append a "tail" of random unused codons. GEVA does not have any such option. AGE supports it optionally, but in has not been enabled in this experiment. The option would not, however, cause wrapping events to occur.

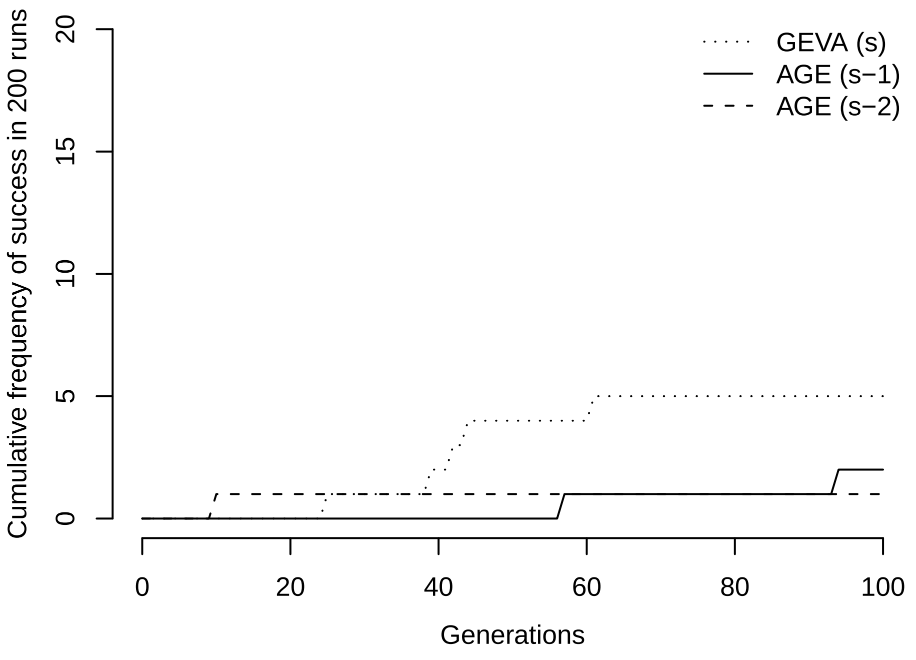


Figure 8.5: Cumulative frequency of success of AGE and GEVA (“sensible” initialisation) in the Santa Fe ant trail. **Note:** GEVA did not use the specified initialisation method properly. (Scale from 0 to 20 out of 200.)

undergo wrapping, which in turn is possible only if the “sensible” initialisation is implemented incorrectly. Additionally, the average tree height reported by GEVA (5.65) is higher than it should be. The minimum tree height for this grammar is 4, the maximum specified height for the initialisation is 6, which should result in one third of the population of tree height 4, one third of tree height at most 5, and one third of tree height at most 6.[6] The average tree height should not therefore be higher than 5.

The better results were, as we can now see, caused by a bug in GEVA, which suggests that the “sensible” initialisation does not work better than random initialisation in conjunction with this form of grammar and the maximum tree height 6. This is why I have repeated the same test with random initialisation.

In this second test (Figure 8.6), where both AGE and GEVA used the same initialisation method, the results of AGE with 8-bit codons and GEVA were in the end of the run significantly better than the previous ones of GEVA, reaching 7 successful runs out of 200. AGE with 31-bit codons had a success rate of 6 runs. The three configurations achieved virtually identical results: without significant differences in most generations including the last one. As in the previous experiment, there is no evidence that the 31-bit codons are of any benefit.

The initial tests have proved that AGE produces results competitive with GEVA. The performance is compared in Figure 8.7. When “sensible” initialisation is used, AGE is approximately 29 times faster then GEVA, when random initialisation is used, it is approximately 26 times faster, regardless of codon size.

[6]This partitioning of the population is violated only if generation of unique individuals is required, and the tree heights do not allow this, but GEVA does not support generation of unique individuals. AGE does support it in the same way it was originally implemented by Koza (1992) for his ramped initialisation, but in has not been enabled in this experiment.

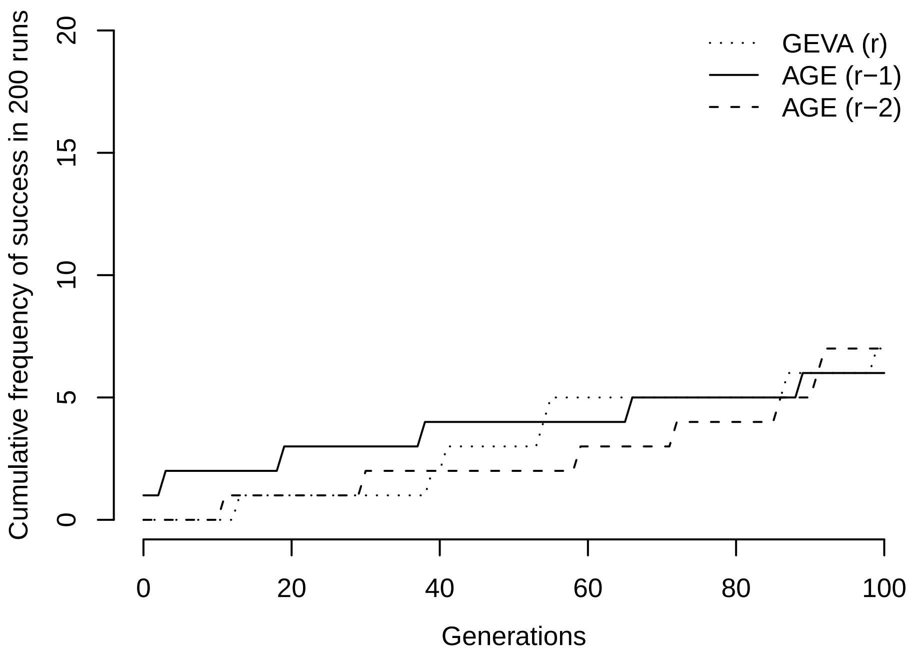


Figure 8.6: Cumulative frequency of success of AGE and GEVA (random initialisation) in the Santa Fe ant trail. (Scale from 0 to 20 out of 200.)

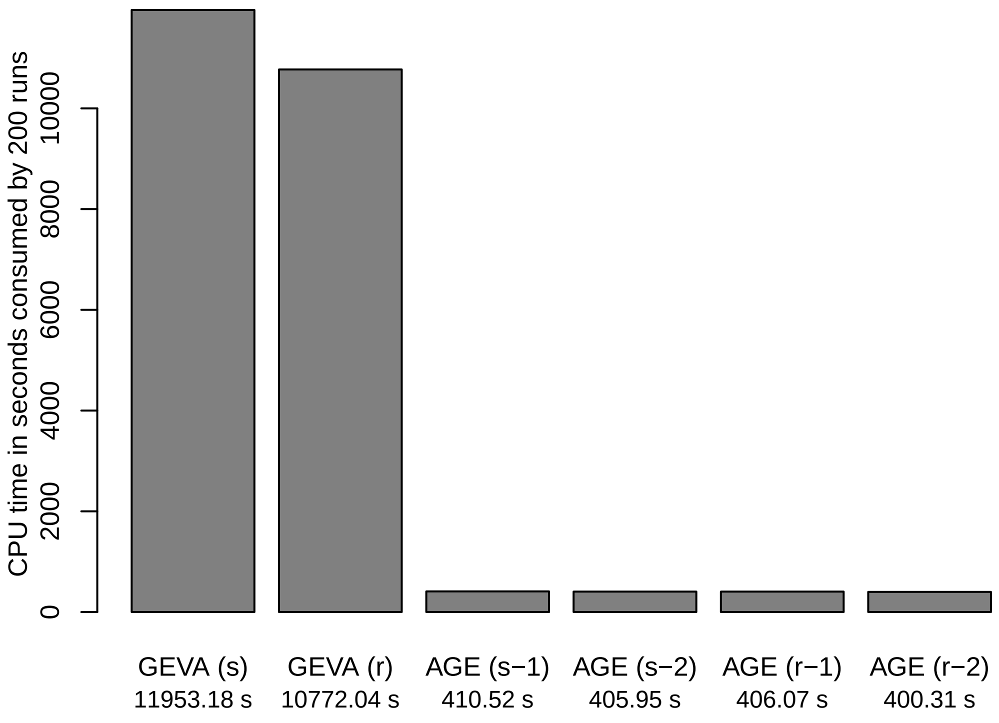


Figure 8.7: Running times of AGE and GEVA for the Santa Fe ant trail.

8.3.3 Further experiments

In the following set of experiments we will explore several adjustments to the initial setup. The experiments will be performed with a population of 500 so that the changes are more visible. Additionally, they will be done with bit-level mutation (1% probability), a steady-state algorithm (1.0 replacement), roulette-wheel selection, linear scaling (factor 8), and standard fitness (food eaten instead of food left). Other parameters will be the same as in *AGE (2)*.

As we are using a larger population, there is no point in comparing the results to the initial ones. Instead we will be interested in comparison of different initialisation methods:

- *AGE (1′)*: random initialisation to 15–25 codons
- *AGE (2′)*: "sensible" initialisation with maximum height 6.
- *AGE (3′)*: ramped initialisation (default settings) with maximum height 6.
- *AGE (4′)*: ramped initialisation with maximum height 6 and uniqueness.

As we can see in Figure 8.8, both the "sensible" and default ramped initialisation had significantly worse results (success rates 35 and 34 out of 200) than the simple random initialisation (success rate 45 out of 200), and the final difference between the two is insignificant. The ramped initialisation with uniqueness, on the other hand, showed a significant improvement (success rate 47 out of 200). This is because it actually generates trees higher than the maximum 6, to provide a wide enough range of unique individuals, as we use the same method for generation of unique individuals as Koza (1992) for his ramped initialisation. In this way it automatically corrects the unnecessarily low requirement for maximum tree height.

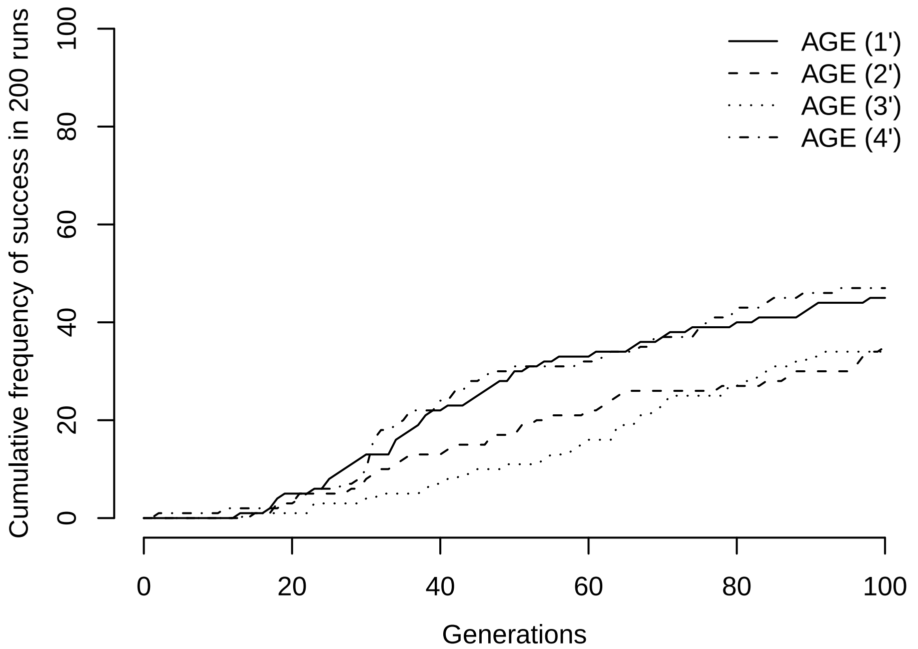


Figure 8.8: Cumulative frequency of success of AGE (alternative parameters) in the Santa Fe ant trail. (Scale from 0 to 100 out of 200.)

The different initialisation methods affected performance only slightly. The running times for (200 runs each) of *AGE (1′)*, *AGE (2′)*, *AGE (3′)*, and *AGE (4′)*

were 1963.09, 1874.33, 1861.94, and 2014.88 seconds respectively.

8.4 Conclusion

AGE has produced competitive results with settings emulating GEVA in both applications. In the symbolic regression experiment it had almost double the success rate of GEVA, in the Santa Fe ant trail experiment it had virtually the same success rate. Surprisingly, it also outperformed GEVA in running time by an order of magnitude. This applies regardless whether AGE used 8-bit or 31-bit codons.

By the subsequent modification of parameters, we have shown the usefulness of additional algorithm elements present in AGE:

- Fast bit-level mutation has better performance than codon-level mutation for the low mutation rates usually used in GE.
- The ramped initialisation (generalised “sensible” initialisation) is no worse than the original “sensible” initialisation. In contrast with GEVA, it is implemented correctly and supports generation of unique individuals, which improved results in our experiment.
- The roulette-wheel selection in conjunction with appropriate scalings achieved better results than the tournament selection used in GEVA.

I have also tried to compare results from the ant trail application with libGE, which implements it in one of its examples. I have been able to build the most recent version of libGE on Mac OS X, which is a fully POSIX-compliant Unix system, but the tested example crashed consistently both with GCC and Lua evaluation. As I have found out, the crashes have been reported on the official libGE forum in 2004 and again in 2006 without any reaction.[7] As of 2009 libGE is no longer maintained.

It would be very valuable to reproduce the results presented by O’Neill and Ryan (2003). The book unfortunately does not provide precise parameters used in the experiments. Throughout its whole text there is not a single mention of any particular selection scheme or scaling. Some of the algorithm elements used in the experiments are named but not defined precisely (such as steady-state evolution). The software used for the experiments is not identified, but it is neither libGE nor GEVA, both of which were developed later.

[7]The official libGE forum topic *mac os x 10.4.5 on Intel Core Duo & PowerPC G5* at `http://bds.ul.ie/libGE/FF/viewtopic.php?t=10`, posts by the users *oneillm* from 2004 and *cretog8* from 2006.

Chapter 9

Conclusion

9.1 Summary

We have described and discussed grammatical evolution, a grammar-based approach to genetic programming introduced by Michael O'Neill (described by O'Neill and Ryan, 2003), in the context of relevant methods of genetic programming. We have examined the existing implementations of grammatical evolution, and developed a new one. This new software framework, *AGE*, attempts to avoid the deficiencies of previous implementations and stresses the reproducibility of results. In the course of designing, implementing and testing AGE, we have pointed out the areas of grammatical evolution that would benefit from further research. The finished software tool has been tested in two benchmark applications. It proved itself capable of delivering competitive results, which are also reproducible and allow for further analysis. Additionally, it outperformed the only other relevant implementation by an order of magnitude in terms of computational time. Several experiments have been conducted to show how to take advantage of the algorithm elements implemented in AGE and improve the results in the benchmark applications.

In the introductory chapter, we have focused on the areas essential for a sound design of a software framework for grammatical evolution: the notion of fitness, and its pivotal role in evolutionary algorithms, the genotype-phenotype distinction as understood in grammatical evolution, identification and clear description of the most widely used algorithm elements.

Following this introduction, we have reviewed the two major existing implementations of GE in Chapter 3. We have noted that there are two approaches to implementing GE: as a module for an existing EA environment, or as a comprehensive framework with direct support for GE. The former path has been taken by libGE, the latter by GEVA. By examining the implementations, we have recognised the risks of the two approaches: clumsiness and poor integration in the case of a GE module; poor documentation and an unclear design rationale in the case of the full-fledged environment. We have also reviewed both existing implementations with respect to the methods and standard algorithm elements presented in the introduction, and described their general characteristics such as the form of output and portability. This has shown their strengths and weaknesses.

In Chapter 4 we have set the goals for AGE, a new implementation of grammatical evolution. We have opted for implementing it as a stand-alone modular framework for evolutionary algorithms with direct support for GE. The most important goals are a clean, comprehensive implementation of standard algorithms, modularity, an adequate documentation, output suitable for further analysis, reproducible results, and consequently also portability.

Both the overall design resulting from these goals and other important design decisions were described in Chapter 5. When planning the implementation of algorithm elements, we have pointed out the possible weak points of some methods used in conjunction with grammatical evolution, namely the "sensible" initialisation (devised by O'Neill and Ryan, 2003), and the mutation operators (bit-level mutation, codon-level mutation, and duplication). Two improvements have been suggested and implemented in AGE: fast bit-level mutation, and a modification to the "sensible" initialisation. We have acknowledged the interactions between the operators, initialisation schemes and the structure of the grammar used for the mapping. We have argued that this area of grammatical evolution is in need of a more thorough research, which is, however, out of scope of this thesis.

The finished software tool has been documented in Chapter 6 and Chapter 7 with references to the previously described methods and techniques.

Finally, in Chapter 8, AGE has been tested in two benchmark applications, and the results have been compared with the only other relevant implementation, GEVA (O'Neill et al., 2008). AGE produced competitive results, and also outperformed GEVA by an order of magnitude in terms of computational time: 10 to 29 times based on application (with the same settings). Further experiments with the two applications have been conducted. The experiments have shown viability of the modifications suggested in Chapter 5.

9.2 Ideas for further research

This thesis and its accompanying software project have fulfilled its goals, but as we have stated in Chapter 1, it is meant to serve as a tool for further experiments and research.

The future research of grammatical evolution, and possibly other grammar-based methods of genetic programming, can focus on two areas that complement each other: modifications to the mapping, and adaptation of existing methods to the needs of GE of evolutionary algorithms or development of new, more suitable ones.

Grammars

Obviously, the choice of the grammar delimits the search space. Moreover, equivalent grammars differing only in their form encode a preference for certain structures. (Examples are provided in Section 2.5.1 and Section 8.3.) This has been realised by O'Neill and Ryan (2003), who suggest a dynamic co-evolution of a grammar for further research.

When examining the "sensible" initialisation method in Section 5.2 we have pointed out that converting the grammar to some predefined form can make the mapping, and consequently the operators, behave more predictably. For instance, we could require the right-hand side of each production rule to contain at least one nonterminal. The standard forms (such as Chomsky normal form or Graibach normal form), which were devised for different purposes, may not be useful for grammatical evolution: it clearly is not sufficient that the form is clearly defined, it must have properties useful in conjunction with the mechanisms of the mapping.

The use of predefined grammar forms seems to go in the opposite direction of the dynamic co-evolution, but in fact the two approaches are not mutually exclusive: the conversion to a certain form could be used as a part of initialisation in the evolution of the grammar.

A specific area we have briefly touched in Section 2.5.1 is the evolution of ephemeral constants. Dempsey et al. (2009) have already conducted some research in this area.

Mapping

The mapping process devised for grammatical evolution could be changed radically (some possibilities are suggested by O'Neill and Ryan, 2003, sec. 9.2). There are, however, also ways to affect the mapping indirectly, which leads us again to the structure of the supplied grammar, and also to the codon size. The codon size could be either chosen automatically based on the numbers of productions per nonterminal in the grammar, or it could also be subject to evolution.

Algorithm elements

As we have noted in Section 2.6.5, the effective mutation rate of the bit-level mutation is determined jointly by the codon size, length of the chromosomes, structure of the applied grammar and wrapping. Additionally, as we have said in Section 5.4, both the bit-level mutation and the codon-level mutation tend to have an all-or-nothing change effect on the phenotype, instead of affecting it locally. This leaves a lot of room for research of alternative mutation operators.

The "sensible" initialisation, introduced by O'Neill and Ryan (2003) as an analogue of Koza's ramped half-and-half initialisation, has been shown to be helpful, but as we have recognised in Section 5.2, the algorithm has been designed somewhat arbitrarily. As demonstrated in Chapter 8, the special treatment of recursive productions is indeed superfluous. An improved initialisation operator could take into account the structure of the grammar in a more effective way.

We have also briefly mentioned fitness sharing in Section 2.6.4, a technique that could be adapted for use in grammatical evolution.

O'Neill and Ryan (2003, sec. 9.2) suggest other, even more significant changes, such as employment of "competent GAs", which have improved scaling characteristics, or alternative search strategies.

Evolutionary dynamics

The need for a more thorough investigation of the dynamics of the evolutionary process has been already acknowledged by O'Neill and Ryan (2003, sec. 9.2).

Applications and experiments

As we have said in Section 8.4, it would be very valuable if the results presented by O'Neill and Ryan (2003) could be reproduced. I am unaware of any other direct comparison of results achieved by GE and other methods of genetic programming, unfortunately the authors did not mention many important details about the experiments.

The experiments presented in this thesis serve primarily for demonstration of AGE and its comparison with GEVA. It would be useful both to compare results in benchmark applications with methods other than GE and develop real-world applications that would fully exploit the ability of GE to evolve programs in any language.

Chapter 10

Changes to the Original Text

This text was originally written as a bachelor thesis, which was successfully defended in September 2009. Since then it has been tweaked a bit. None of the changes are substantial, but I nevertheless provide a full list of them in this chapter:

5 April 2010, version 1.0.1

- Formatting: added hyperlink highlights for better onscreen reading (not visible when printed), typography tweaks.
- Title pages: a version number added.
- Page 5: licence terms added.
- Page 12: a reference to Chapter 10 (this chapter) added.
- Page 20: erroneous "line Section 7" replaced with correct "line 7".
- Page 47: NetBSD mentioned among the supported platforms.
- Page 49: variable assignments (`INSTALL_PREFIX=`... and `EXTRAFLAGS=`...) swapped with targets (`install`, `ansi`, and `all`) in `make` invocations for better clarity. (While any order works both in BSD and GNU make, the BSD make documentation specifies that variables go first, which is also more intuitive.)
- Page 49: the first sentence of the *Manual build procedure* section rephrased.
- Page 54: the numbers in the paragraph referring to Listing 6.2 and Listing 6.3 (page 55) adjusted to reflect the updated listings (see below). No change in wording made.
- Page 55: Listing 6.2 and Listing 6.3 updated to transcribe actual output from the final version of AGE (1.0). The original version mistakenly contained outputs from an earlier development version. (The output from versions 1.0 and 1.0.1 is identical.)
- Page 100: a note about licence terms of this text added, copyright date updated.

- Page 102: superfluous comma after “realise” deleted.
- Pages 119–120: Figure 8.8 updated to reflect a bug fix made in AGE version 1.0.1, which affected the results of the *AGE (4′)* configuration (ramped initialisation with uniqueness in the Santa Fe ant trail). The resulting success rate for that configuration (47 out of 200) was explicitly added to the text, and the running time was updated from 1991.57 seconds to 2014.88 seconds (a 1% change). All assertions made in the original version still hold.
- Page 120: a full stop added after “[. . .] used in the experiments”.
- Page 125: Chapter 10 (this chapter) added.

Changes to the software made in version 1.0.1 are documented separately in the `CHANGES` file.

20 July 2009, the original bachelor thesis

The original thesis was not published on the web, but is available in the library of Faculty of Mathematics and Physics, Charles University in Prague. It differs from the text you are reading only as indicated in the list above.

Bibliography

Wolfgang Banzhaf, Frank D. Francone, Robert E. Keller, and Peter Nordin. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. ISBN 1-55860-510-X.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, second edition, 2001.

Ian Dempsey, Michael O'Neill, and Anthony Brabazon. *Foundations in Grammatical Evolution for Dynamic Environments*, volume 194 of *Studies in Computational Intelligence.* Springer, 2009. URL `http://www.springer.com/engineering/book/978-3-642-00313-4`.

Luc Devroye. *Non-Uniform Random Variate Generation.* Springer Verlag, 1st edition, 1986. URL `http://cg.scs.carleton.ca/~luc/rnbookindex.html`.

David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, 1989. ISBN 0201157675.

John R. Koza. *Genetic programming: On the programming of computers by natural selection.* MIT Press, Cambridge, Mass., 1992. ISBN 0-262-11170-5.

Richard Lewontin. The genotype/phenotype distinction. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* 2008. URL `http://plato.stanford.edu/archives/fall2008/entries/genotype-phenotype/`.

Miguel Nicolau and Darwin Slattery. *LibGE Documentation.* Biocomputing-Developmental Systems Centre, University of Limerick, version 0.27alpha1 edition, 2006. URL `http://bds.ul.ie/libGE/documentation.html`.

Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Springer, 1st edition, 2003.

Michael O'Neill, Erik Hemberg, Conor Gilligan, Eliott Bartley, James McDermott, and Anthony Brabazon. *GEVA – Grammatical Evolution in Java (v 1.0).* Technical report, UCD School of Computer Science and Informatics, 2008. URL `http://www.csi.ucd.ie/files/ucd-csi-2008-09.pdf`.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. URL `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).