GECO: A CLOS-based Framework for Prototyping Genetic Algorithms

— Version 2.0 —

George P. W. Williams, Jr.
george@hsvaic.hv.boeing.com

November 27, 1993

Abstract

GECO (Genetic Evolution through Combination of Objects) is an extensible, object-oriented framework for prototyping genetic algorithms in Common Lisp. GECO makes extensive use of CLOS, the Common Lisp Object System, to implement its functionality. The abstractions provided by the classes have been chosen with the intent both of being easily understandable to anyone familiar with the paradigm of genetic algorithms, and of providing the algorithm developer with the ability to customize all aspects of its operation. This paper provides a description of GECO including its internal structure, and presents a simple example genetic algorithm implemented on top of GECO. The author has made the implementation freely available via the Internet.

Contents

List of Figures

Chapter 1

Introduction

1.1 Purpose

Genetic algorithms (GAs) [Hol92, Gol89] covers a broad category of robust adaptive techniques. They have been used to solve an increasingly large variety of difficult problems, including parameter optimization, combinatorial optimization, rule learning, pattern recognition, automatic programming, adaptive control, *etc.*

The GA research community has produced a number of reusable GA implementations [Gre84a, Gre84b, Gol82, WK88, Spe91], but many of them implement a specific approach to GAs which their authors prefer, or which reflects their research orientations. Though these implementations can be adapted to solving different problems, the amount of effort required to customize them generally increases dramatically as the proposed changes vary from the original visions of their implementors.

Developing a GA-based application often requires a great deal of experimenting — trying variations on the genetic operators, the selection algorithm, establishing various parameter settings, *etc.* Regardless of what language is used for implementing the final system, it often makes a great deal of sense to develop a prototype first, establishing the best kind of GA for the application and gaining a good understanding of the effects of varying each of the parameters, before proceeding to create the production implementation.

GECO (Genetic Evolution through Combination of Objects) is an attempt to construct a more flexible framework for prototyping GAs. It makes extensive use of object-oriented implementation techniques to provide this flexibility, and it was designed from the outset with the goal of being easily customized and extended.

Although GECO is intended for developing more-or-less classical GAs, it appears to be sufficiently flexible to be used for prototyping some related classes of algorithms: Learning

Classifier Systems (LCSs) [HR78, HHNT87], Genetic Programming (GP) [Koz92], Evolution Strategies (ESs) [BHS91], and Evolutionary Programming (EP) [MJ91].

1.2 State of the System

GECO is a work in progress. It is still in its infancy, and does not provide all the features required for it to be considered a complete GA toolkit. But where it lacks functionality, it can be easily extended to implement new capabilities. Furthermore, as it continues to grow, its structure will certainly continue to evolve to make it even more adaptable. Much like other evolutionary processes, I hope to improve GECO as I learn through feedback from its community of users.

GECO is copyrighted free software. The complete source code is available directly from the author, and from a number of anonymous FTP archives on the Internet.

This version of the GECO documentation refers to version 2.0 of the software.

1.3 Why Lisp?

Since GAs are computationally expensive, most implementations are in 'conventional' programming languages such as `C` "for efficiency." Lisp is a much more convenient language for prototyping, since almost all implementations are highly interactive and provide a great deal of support for the development process. Common Lisp is becoming the standard industrial strength dialect in the United States, and it has many high quality implementations on a wide variety of platforms. Furthermore, the compilers are capable of producing code which often rivals 'conventional' language implementations for speed and efficiency. Common Lisp is also one of the most portable languages available today — in many ways more portable than `C` or Ada. The relatively recent addition of the Common Lisp Object System (`CLOS`) to the Common Lisp language [Ste90, Kee89] has provided a full-featured object-oriented extension to the language, which greatly enhances it's capability for prototyping and writing extensible libraries.

1.4 Notational Conventions

This document follows several notational conventions for the sake of conciseness, and to assist the reader in understanding usage in context.

1.4.1 Terms and Concepts

Most new (and some common) terms and concepts which are used in this document are set in sans serif type when they are introduced and/or defined, or when it seems important to emphasize that the word or phrase has a specific meaning in context. In addition, the location in the document where the concept is explained or defined appears in the index.

1.4.2 Source Code

All source code is set in `typewriter` style type, whether it appears in the body of the text, or set off from the body of the text as an example. This includes names of Common Lisp or GECO-defined entities, such as names of functions, classes, *etc.*, which will also be set in `typewriter` style type.

1.4.3 File Names, Network Paths, ...

All such references will be set in `typewriter` style type.

1.4.4 Descriptions of GECO-defined Entities

The syntactic descriptions of functions, methods, variables, classes, and other definitions are presented in a distinctive format, similar to that used in Guy Steele's *Common Lisp: The Language* [Ste90], and many other similar documents since. This stylized description is generally followed by narrative body text explaining the intended usage and semantics. References in the body text to components of the syntactic description will appear in the same type style in which the component appears in the syntactic description.

The first line of one of these syntactic descriptions is always signalled by a $\Rightarrow$ symbol appearing in the left margin; this line is referred to below as a flag line, since the symbol in the margin helps to *flag* the readers attention. This flag line specifies the name of the definition against the left margin, with the type of definition in italics and brackets against the right margin. For example:

$\Rightarrow$ `example-variable` [*Variable*]

For functions, generic functions, and macros, any arguments appear on indented lines immediately following the flag line, with the argument names set in *slanted* type. Common Lisp lambda-list keywords (*e.g.*, `&optional`, `&key`) are set in `typewriter` style type, since they are literal language constructs in the definition (though they do not appear when the form is used). In addition, any `&optional`, `&key`, or `&rest` arguments which have default values are shown parenthesized, with the default value specification shown in `typewriter` style type, since it is a Common Lisp literal.

⇒ `example-function` [*Generic Function*]
  (*arg1 arg2* `&optional` *arg3*)

For methods, the second and subsequent lines specify the argument list, but also shows the specialization of arguments. *I.e.*, specialized arguments are shown in parentheses, with the specializing class following the argument, and in `typewriter` style type, since it is a literal reference to a class.

⇒ `example-function` [*Primary Method*]
  ((*arg1* `class1`) *arg2* `&optional` (*arg3* `'default-value`))

Variables, constants, and slots are specified in a similar manner. However since these entities may have default or initial values, such values are shown on indented lines immediately following the flag line. Similarly, class description flag lines may be followed by indented lines identifying any superclasses of the class.

⇒ `example-slot` [*Slot*]
  `default-slot-value`

Slot descriptions always follow the description of the class on which they are directly defined (descriptions of inherited slots are not repeated). Accessor functions and initargs for slots (if any) are specified on flag lines immediately following the descriptions of their slots.

⇒ `example-slot-accessor` [*Accessor*]

⇒ `:example-initarg` [*Initarg*]

1.4.5 The Index

All of GECO's definitions are doubly indexed, by both name and type of definition (*e.g.*, function, class, slot, *etc.*). In addition, all references to GECO-defined entities in the body of the text are indexed. To aid the user in finding the definition in a potential forest of references, the page number of the definition(s) appears in *italics*. Note that in some cases, there may be more than one definition of a method, and the generic function definition is generally repeated in these cases as well.

1.5 Acknowledgments

I want to thank Randy Fennel and Al Underbrink for their help in beta-testing the software, and for their comments on early drafts of this documentation. Of course I am solely responsible for any errors which surely remain. I also want to thank John Koza for kindly permitting the reproduction of his implementation of the Park-Miller randomizer, Kate Juliff for inspiring the multi-chromosome features, and Larry Yaeger for publishing the `C` implementation of gray code translation after which my implementation (section 3.8.3) is patterned.

Chapter 2

GECO Concepts and Structure

Geco predefines classes and methods to simplify the process of constructing a GA for a specific application. This section discusses geco's approach to implementing GAs through a discussion of the classes, methods, and related functions it implements.

2.1 Overview of Geco Classes

Geco is an object-oriented library which implements an environment primarily in the form of classes and methods. Geco's classes are based on the natural concepts which are part of the genetic evolutionary paradigm. The principle classes form a hierarchy of *abstractions* (*not* a *class* hierarchy) which parallel the natural concepts of genetic evolution. Objects which are higher in this abstraction hierarchy 'contain' objects which are lower in the hierarchy (see Figure 2.1). We (GA developers) build on these classes and methods to describe and implement our GAs. The terminology and concepts used within geco may differ slightly from other conventional usage, but they hopefully are internally consistent, and should be intuitive to a reader who is conversant with the concepts employed by GAs.

Here are the principle classes in the geco hierarchy of genetic abstractions, starting from the top:

Ecosystem A combination of the population undergoing evolution, and the genetic plan which controls the evolution.

Population GAs evolve populations of organisms. The current population at any time is the set of organisms which can interact with one another to produce new organisms.

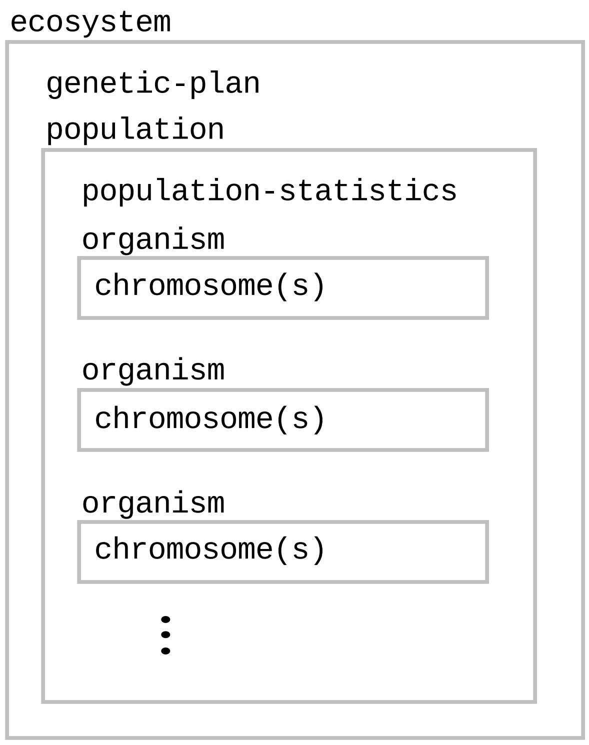


Figure 2.1: GECO's classes form an abstraction hierarchy

Organism An organism combines all the related information about a single structure in the search space being explored by the GA. An organism is a member of a population, generally has a coded genetic description (its genotype), and interacts with its environment as the individual (its phenotype) coded by its genotype. Each organism also has a score, which is used to establish an organism's relative value toward solving the specific problem posed for solution by the GA.

Chromosome A structured component of an organism's genotype, which generally is the unit which is operated upon by genetic operators. Many GAs use only a single chromosome per organism, but sometimes there are reasons to use more than one. Each chromosome is generally composed of a vector of loci (sites), each of which may take on one of a set of values (the alleles for that locus). The arity of each locus is typically the same for all loci of a single chromosome, but GECOis sufficiently general that arity may be a function of locus. Genetic interpretation is normally (but not necessarily) a function of position on the chromosome.

> Terminological aside: a gene might be defined as a functional or operational unit by which genetic information is transferred from parent to offspring, which may consist of one or more alleles from one or more loci, which may or may not be contiguous on a chromosome. The exact definitions of genes and alleles in the context of GAs has historically been rather vague. Only recently have attempts been made to define them formally; see [Rad92a, Rad92b].

There are also some other important classes which, though they aren't part of the geneticly-based abstraction hierarchy, play important roles in GECO's operation.

Genetic Plan The overall strategy which determines how an ecosystem regenerates, *i.e.*, how new organisms are created from older organisms. This generally includes the overall scheme for selection of organisms for reproduction, replacement, and manipulation by genetic operators. Methods defined on this class will generally determine how a particular GA implementation differs from the canonical GA described by Holland. An instance of this class is a component of each ecosystem.

Population Statistics Information accumulated about (at least) the scores of the members of a population, used for normalizing the scores across the population, *etc.* An instance of this class is a component of each population.

In addition, there are other classes in GECO. Some of these classes specialize on the classes above, while others serve auxiliary purposes.

Organism Phenotype Mixin An abstract class (one not intended to be instantiated), which may be included as a parent class in a subclass of the `organism` class.[1] This class adds a `phenotype` slot to the subclass, along with relevant behavior, when a phenotype must actually be created from the genotype. Since this is not always necessary, the `phenotype` slot has been abstracted out of the `organism` class.

Maximizing & Minimizing Score Mixins Two abstract classes, one of which should be included as one of the superclasses of an application's population class. That is, an instantiable subclass of `population` must also be a subclass of one of these classes (using multiple inheritance). The mixin you choose to include in your population class will determine whether GECO tries to maximize or minimize the organisms' scores in the population being evolved.

Generational Population A subclass of `population` which provides explicit support for the standard generational style of GA. Eventually GECO may contains support for other styles of GA, possibly including parallel populations, steady-state populations, *etc.* This class (or a future alternative) should be included as one of the superclasses of an application's population class.

Binary Chromosome A special kind of chromosome — When each locus of a chromosome may only take on one of two alleles, then each of the loci are binary, and the chromosome which they compose is binary. It is very common for a GA to require only binary chromosomes, and so GECO provides support for this special case.

Sequence Chromosome Another special kind of chromosome — When each locus of the chromosome is treated as a unique item of a sequence, and the chromosome itself specifies a permutation of the sequence. This is another common kind of chromosome, used for applications such as the Traveling Sales-rep Problem (TSP).

Gray Code Translation A special translation table for converting to and from gray coded representations of a specific number of bits. Some applications of GAs using binary chromosomes work better if the genetic coding scheme for some parameters is a gray code.

[1]A mixin class is a class which is *mixed in* with other classes to collectively form the set of parent classes of a new class. A mixin class is almost always an abstract class.

2.2 Basic Flow of Control

Assuming that the appropriate definitions have been made to extend GECO for a specific GA implementation, the basic operation of a typical GA is as follows:

Initialization: Make an instance of the ecosystem class or subclass which will be used for the GA. GECO then automatically creates instances of the appropriate classes for the genetic plan and the population. Creating the initial population instance in turn causes the creation of the initial organism instances which belong to the population, each of which is initialized with random chromosomes of the appropriate classes (see Figure 2.2).

Evolution: Invoking `evolve` on the ecosystem causes GECO to evolve the population (see Figure 2.3). This consists of repeating the following steps:

- Evaluate each of the organisms in the current population, recording a score for each one.
- Calculate population statistics, normalized scores for each organism, and normalized population statistics.
- Determine if the GAs termination condition has been met. If it has, then terminate. Otherwise:
  - Regenerate the population. This is where most of the customizing is done for a new GA in GECO. This typically includes selecting members of the previous population to participate in creating the members of the new population. GECO provides a number of predefined functions for performing the selection and for creating the new population based on members of the previous one, *e.g.*, via reproduction, and various kinds of crossover and mutation.
  - Recursively[2] evolve the result.

When supplied with the appropriate information, GECO can perform much of the bookkeeping, initialization, and control automatically. This is made possible by the built-in links between objects which are built upon the GECO classes (see Figure 2.4).

[2]This recursive invocation could lead to 'stack overflow' or similar error conditions in many languages. In Lisp, this particular kind of recursion (called tail recursion) is a special case which can be recognized by the compiler and implemented (very efficiently) as simple iteration. The result is an implementation which is concise, clear, and efficient. For those implementations which do not provide this optimization, an equivalent iterative definition is provided, which can be selected using conditional compilation options specified in the `GECO.system` file. See the comments in that file for details.

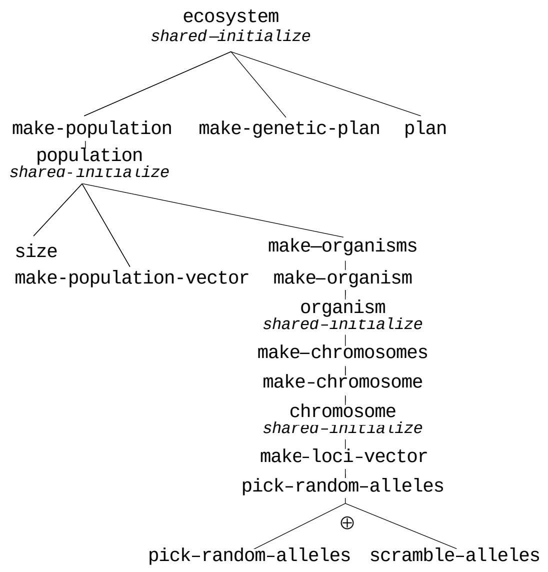


Figure 2.2: Call hierarchy for initialization of GECO's principle structures

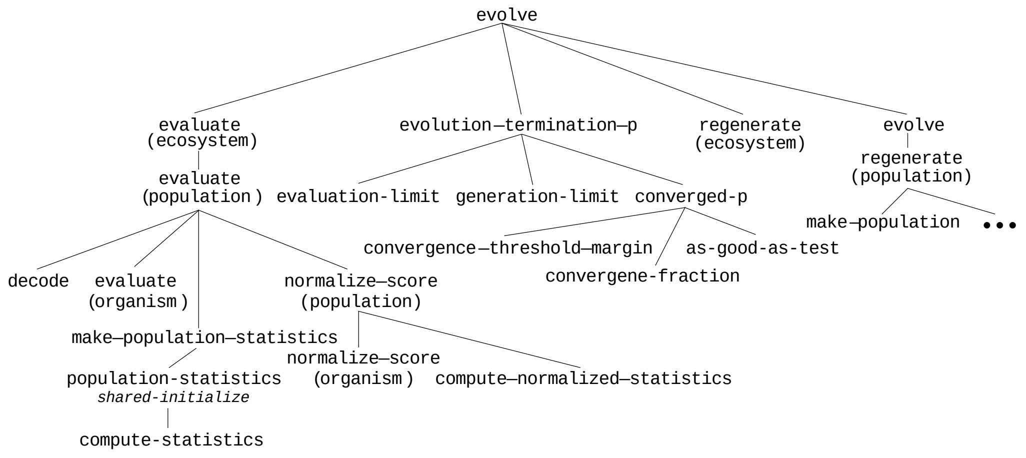
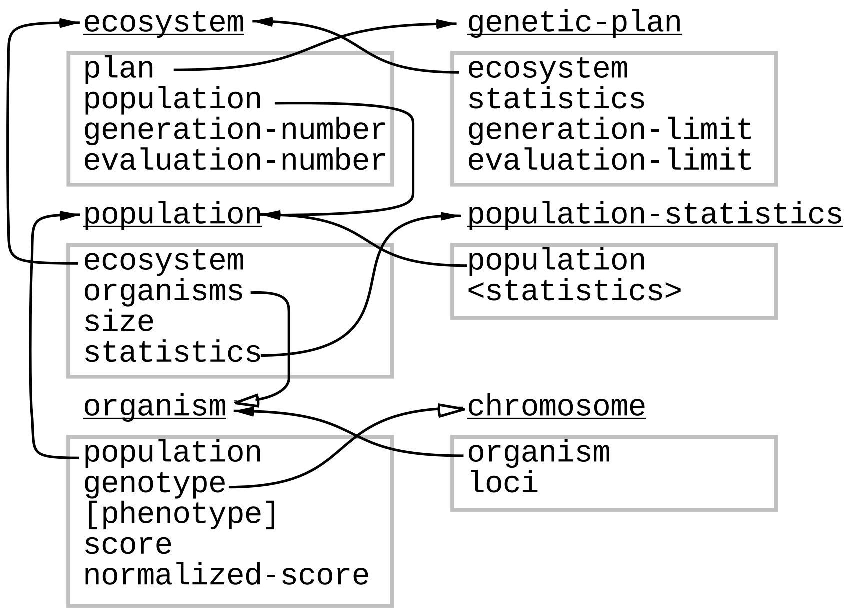


Figure 2.3: Call hierarchy for GECO's evolutionary processing



Solid head arrows indicate a one-to-one link;
hollow head arrows indicate a one-to-many link.

Figure 2.4: Interrelationships between GECO objects

Chapter 3

Details of GECO Classes and Functionality

This chapter provides a more detailed discussion of each of GECO's classes, and the functionality implemented by their methods. This functionality includes both the state retained by instances of each class (their *slots*), and the functions (both generic and otherwise) which operate on those instances.

Even with all the functionality which GECO implements, it will still be necessary to define some things which are specific to your application. Generally this will be done by specializing GECO's classes (*i.e.*, defining some subclasses of GECO's builtin classes), and adding a few method definitions to override and/or extend some of GECO's default behaviors.

Terminology Notes:

- In the material which follows, a statement which refers to 'an instance of a *class-name* class' means that the instance is of the class *class-name* or one of it's subclasses. If the intent is to restrict the instance to being of the named class, excluding subclasses, the wording will be of the form 'an instance of the *class-name* class.'
- In the descriptions of the methods, it will often be necessary to distinguish between a generic function, a method for the generic function, and the specific method supplied by GECO. A generic function and a method (as specialized in the flag line above the description) are both parts of a functional protocol which GECO expects to be honored. The description of the GECO-supplied method specifies hot the GECO-supplied method implements (fulfills) the requirements of this protocol. GECOmay define multiple methods (specialized for different classes) to implement the generic function protocol for different classes.

For each class, the following sections will present the slots which are present in instances of the class (*i.e.*, the values stored with each instance) and the functionality which has been defined for use with instances of the class (and its subclasses). Generally, GAs implemented with GECO will not instantiate these classes. Instead, it will be more common to define subclasses which extend these classes (via added slots and methods) and specialize them (by overriding and/or extending inherited methods).

3.1 The Ecosystem Class

An `ecosystem` is the highest level abstraction in a GECO implementation. It is also the handle for manipulating a particular run of a GA. Since there may be more than one instance of an `ecosystem` in existence at one time, it is possible to use GECO to create applications which use more than one GA at the same time. The individual GAs may be competing, working on separate aspects of the same problem, or they may be completely independent.

⇒ `ecosystem` [*Class*]

Instance Allocated Slots

⇒ `population` [*Slot*]
⇒ `population` [*Accessor*]

An instance of a `population` class. The population of an ecosystem is the set of organisms which are being evolved.

⇒ `generation-number` [*Slot*]
    `0`
⇒ `generation-number` [*Accessor*]

An integer, initially 0, which is incremented each time the population enters a new *generation*. A new generation is created each time the `evolve` function is invoked on an `ecosystem` instance (including `evolve`'s self-invocations).

⇒ `evaluation-number` [*Slot*]
    `0`
⇒ `evaluation-number` [*Accessor*]

An integer, initially 0, which counts the number of times the `evaluate` function is applied to an `organism` instance.

⇒ `plan` [*Slot*]

⇒ `plan` [*Accessor*]

An instance of a `genetic-plan` class.

The number of generations and evaluations are tracked by GECO so that the GA can be terminated based on the number of generations or evaluations exceeding some specific maximum limits, specified by the GA implementor. These limits are among the slots of the class `genetic-plan`.

The `population` and `plan` are distinguished from the `ecosystem` so that their classes may be specialized independently. Thus an instance of a single `population` class may be manipulated using different `plans`, while instances of a single `plan` may be used with different `populations`.

Instance Creation and Initialization

The `ecosystem` instance initialization has been extended to support the following additional initargs:

⇒ `:plan-class` [*Initarg*]

Provide the class for the `genetic-plan` to be used by the `ecosystem`.

⇒ `:pop-class` [*Initarg*]

Provide the class for the `population` instances to be created by the `ecosystem`.

⇒ `:pop-size` [*Initarg*]

Specifies the size to be used when the `ecosystem` creates `population` instances.

⇒ `:generation-limit` [*Initarg*]

Specifies the maximum number of generations which the `ecosystem` will be allowed to evolve.

⇒ `:evaluation-limit` [*Initarg*]

Specifies the maximum number of evaluations which the `ecosystem` will be allowed to perform.

No special functions for the creation of `ecosystem` instances have been defined, since `make-instance` and the standard CLOS protocol it follows provide all the necessary functionality.

The initialization for `ecosystem` instances has been extended to provide for the automatic creation and initialization of the `population` and `plan` instances and slots. The `make-population` and `make-genetic-plan` generic functions (described next) are used to support customization of these actions. The call to `make-population` passes a value of `t` for the `:random` keyword argument, causing the initial population to be initialized to random organisms. If `:generation-limit` and `:evaluation-limit` are specified, the `generation-limit` and `evaluation-limit` slots in the `plan` instance are also initialized.

Specialized Methods

⇒ `make-population` [*Generic Function*]
ecosystem population-class `&key` *:size :random*

⇒ `make-population` [*Primary Method*]
(ecosystem `ecosystem`*) population-class* `&key` *:size :random*

This function provides an abstract interface to creation of the `population` of *ecosystem*. The primary GECO-supplied method invokes `make-instance` on the class *population-class*, passing the *:size* argument, which determines the population size, and the *:random* argument, so that the population can be created with random organisms (intended for creation of the initial population).

⇒ `make-genetic-plan` [*Generic Function*]
ecosystem genetic-plan-class

⇒ `make-genetic-plan` [*Primary Method*]
(ecosystem `ecosystem`*) genetic-plan-class*

This function provides an abstract interface to creation of the `genetic-plan` instance for *ecosystem*. The GECO-supplied primary method invokes `make-instance` on *genetic-plan-class*, and also supplies *ecosystem* as the `:ecosystem` keyword argument so that the plan can be linked to the ecosystem (and vice-versa).

⇒ `evolve` *[Generic Function]*
  ecosystem

⇒ `evolve` *[Primary Method]*
  (ecosystem `ecosystem`*)*

This is the principle function which will be used by GA developers to invoke their algorithm. The GECO-supplied primary method calls `evaluate` on *ecosystem*, and if the termination condition has not been reached (see `evolution-termination-p`), creates a new generation of its `population` via the `regenerate` function, and recurses to evolve some more[1].

⇒ `evaluate` *[Generic Function]*
  thing genetic-plan

⇒ `evaluate` *[Primary Method]*
  (ecosystem `ecosystem`*) genetic-plan*

The purpose of this function is to cause *thing* to be evaluated according to the specified genetic plan. The GECO-supplied primary method for `ecosystem` instances evaluates *ecosystem* by calling `evaluate` on its `population` with *genetic-plan*. (Also see the `evaluate` method specialized for the class `population`, on page 23.)

3.2 The Population Class

A population is the most global structure upon which a GA operates. Although genetic operators are applied to the members (*organisms* in GECO's terminology, though they are often called *individuals*) of a population, it is at the level of the population that the GA is really working.

⇒ `population` *[Class]*

Instances of `population` classes collect all the organisms of a generation.

Instance Allocated Slots

⇒ `ecosystem` *[Slot]*
⇒ `:ecosystem` *[Initarg]*
⇒ `ecosystem` *[Accessor]*

Provides a link back to the ecosystem to which the population belongs.

[1]See the discussion in the footnote on page 13

⇒ `organisms` [*Slot*]

⇒ `organisms` [*Accessor*]

A vector, which contains all the organisms in the population.

⇒ `size` [*Slot*]

`nil`

⇒ `:size` [*Initarg*]

⇒ `size` [*Accessor*]

Either `nil` or an integer, which indicates the size of the population, *i.e.*, the size of the vector in the `organisms` slot. When `nil`, the organism vector will not be created automatically.

⇒ `statistics` [*Slot*]

⇒ `:statistics` [*Initarg*]

⇒ `statistics` [*Accessor*]

An instance of a `population-statistics` class, which holds statistics GECO needs for the population. The `population-statistics` class is distinct from the `population` so that their classes may be specialized independently.

Instance Creation and Initialization

The generic function `make-population` (see page 19) is the GECO interface for creation of `population` instances.

The initialization for instances of `population` has been extended to provide for automatic creation and initialization of the `organisms` vector. The functions `make-organisms-vector` and `make-organisms` are used to permit customization of these initialization actions; `make-organisms-vector` is called when the `size` slot has a non-`nil` value, and `make-organisms` is called only when both `size` and `:random` (below) have non-`nil` values. It is the responsibility of the `genetic plan` to create the organisms after the initial generation.

The population instance initialization has been extended to support the following additional initarg:

⇒ `:random` [*Initarg*]

The value of this keyword is passed to `make-organisms`, and is intended to support automatic initialization of the initial population to random organisms.

Specialized Methods

Note that most (if not all) of the generic functions in Section 3.10, Selection Methods, have methods which are specialized on the `population` class.

⇒ `make-organisms-vector` [*Generic Function*]
population size

⇒ `make-organisms-vector` [*Primary Method*]
(population `population`*) size*

This function provides an abstract interface to creation of the population's organisms vector (the vector which holds *population*'s organisms). The *size* argument determines the size of the vector. The GECO-supplied primary method uses the Common Lisp function `make-array` to create an array of the specified size.

⇒ `make-organisms` [*Generic Function*]
population `&key` *:random*

⇒ `make-organisms` [*Primary Method*]
(population `population`*)* `&key` *:random*

This function provides an abstract interface to creation of the organisms in *population*'s organisms vector. The *:random* argument, when non-`nil`, causes all the new organisms to be random (*i.e.*, have randomly chosen chromosomes). The GECO-supplied primary method invokes `make-organism` for each position in the organisms vector. The *:random* argument is passed to each call to `make-organism`.

⇒ `make-organism` [*Generic Function*]
population `&key` *:random :no-chromosome*

⇒ `make-organism` [*Primary Method*]
(population `population`*)* `&key` *:random :no-chromosome*

This function provides an abstract interface to creation of a single organism based on the `organism-class` of *population*. The *:random* argument, when non-`nil`, causes the new organism to be random (*i.e.*, have randomly chosen chromosomes). The *:no-chromosome* argument, when non-`nil`, causes the organism to be created without chromosomes, avoiding wasted work when the chromosomes will be supplied by other mechanisms, *e.g.*, genetic operators. The GECO-supplied primary method passes *population* to the call to `make-instance` so that the organism can have a back-link to the population to which it belongs. The *:random* and *:no-chromosomes* arguments are passed to `make-instance`.

⇒ `organism-class` [*Generic Function*]
population

This function returns the class to be used to create organisms which will become members of *population*. The GA developer *must implement the primary method* for all subclasses of the class `population`. GECO does not provide a default primary method specialized on the `population` class.[2]

⇒ `evaluate` [*Generic Function*]
thing genetic-plan

⇒ `evaluate` [*Primary Method*]
(population `population`*) (genetic-plan* `genetic-plan`*)*

This function evaluates *thing* according to *genetic-plan*. This method assures that each organism in *population* is evaluated. The GECO-supplied primary method only calls `evaluate` on an organism if the organism doesn't already have a `score` in its `score` slot. After *population* has been evaluated, `normalize-score` and `make-population-statistics` are called to assure that normalized scores and statistics have been computed for the population.

⇒ `make-population-statistics` [*Generic Function*]
population

⇒ `make-population-statistics` [*Primary Method*]
(population `population`*)*

This function provides an abstract interface to creation of the `population-statistics` instance for *population*, based on the `population-statistics-class` of *population*. The GECO-supplied primary method passes *population* to `make-instance` so that the instance can have a back-link to the population to which it belongs.

⇒ `compute-statistics` [*Generic Function*]
population

⇒ `compute-statistics` [*Primary Method*]
(population `population`*)*

This function provides an abstract interface for computing statistics for *population*. This method provieds a place for a population class to provide for customization of statistics computation. The GECO-supplied primary method simply calls `compute-statistics` on the statistics instance of *population*. (Also see the description of `compute-statistics` specialized on the class `population-statistics` on page 66.)

[2]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

⇒ `compute-binary-allele-statistics` [*Generic Function*]
population

⇒ `compute-binary-allele-statistics` [*Primary Method*]
(*population* `population`)

This function returns a list of vectors (one per binary chromosome in the organisms of *population*) of counts (`fixnums`), by locus, of non-zero alleles. For example, if the organisms in a population contain c binary chromosome (and any number of non-binary chromosomes), and each binary chromosome contains b loci, then this function will return a list containing c vectors of b fixnums. Each `fixnum` in the returned vectors is a count of non-zero alleles in the entire population at the locus whose index corresponds to the index into the c^{th} vector of counts. *I.e.*, if the third count in the first vector is 7, then the entire population contains 7 non-zero alleles in locus 3 of the first binary chromosome of each organism.

⇒ `normalize-score` [*Generic Function*]
thing statistics genetic-plan

⇒ `normalize-score` [*Primary Method*]
(*population* `population`) (*statistics* `population-statistics`)
(*genetic-plan* `genetic-plan`)

This function computes the normalized `score`(s) for *thing*. This method computes the normalized scores for all organisms in *population*. The GECO-supplied primary method for `population` invokes `normalize-score` (see page 35) for each organism in *population*, according to the *genetic-plan*, and updates *population-statistics* with normalized values using the function `compute-normalized-statistics`.

There are a number of different ways to normalize the scores. With some plans and evaluation functions, it may not even be necessary, though beware that the score should always be ≥ 0 (see Chapter 4 of [Gol89], under the sections on Scaling Mechanisms and Ranking Procedures).

⇒ `population-statistics-class` [*Generic Function*]
population

⇒ `population-statistics-class` [*Primary Method*]
(*population* `population`)

This function returns the population-statistics class which will be used for *population*. The GECO-supplied primary method specialized for the `population` class returns `population-statistics`.

⇒ `converged-p` [*Generic Function*]
    population

⇒ `converged-p` [*Primary Method*]
    (*population* `population`)

This function is a predicate which indicates whether *population* has converged, which is useful as a termination condition. The GECO-supplied primary method defines convergence as either of the following:

1. All organisms in *population* have the same `score`; or
2. At least a portion of *population* (specified by the `convergence-fraction` function) has a `normalized-score` which is *as good as* the value specified by the `convergence-threshold-margin` function.

Note that this allows GECO to either *maximize* or *minimize* scores. The mechanism for determining whether GECO maximizes or minimizes, and hence how it determines *as good as* or *better than*, is determined by mixing one of two classes with the population class used by the GA. These mixin classes are described below, in section 3.4.

3.3 Subclasses of Population

⇒ `generational-population` [*Class*]
    population

This class is a subclass of `population` which provides explicit support for the 'standard' generational style of GA. The class has no slots, but methods described elsewhere specialize on this class (see `regenerate`, page 58).

Eventually GECO may contains support for other styles of population handling, possibly including parallel sub-populations, steady-state populations, *etc.*

Instance Creation and Initialization

The generic function `make-population` (see page 19) is the GECO interface for creation of instances of `population` and its subclasses.

3.4 Population Mixin Classes

⇒ `maximizing-score-mixin` *[Class]*

⇒ `minimizing-score-mixin` *[Class]*

Neither of these classes has any slots or has special provisions for instance creation or initialization.

Specialized Methods

Both classes implement methods for the following generic functions:

⇒ `maximizing-p` *[Generic Function]*
population

⇒ `maximizing-p` *[Primary Method]*
(population `maximizing-score-mixin`*)*

⇒ `maximizing-p` *[Primary Method]*
(population `minimizing-score-mixin`*)*

⇒ `minimizing-p` *[Generic Function]*
population

⇒ `minimizing-p` *[Primary Method]*
(population `maximizing-score-mixin`*)*

⇒ `minimizing-p` *[Primary Method]*
(population `minimizing-score-mixin`*)*

These functions permit algorithms to efficiently determine whether the *population* is minimizing or maximizing. The GECO-supplied methods return either `t` or `nil` as appropriate for their class.

⇒ `convergence-fraction` [*Generic Function*]
population

⇒ `convergence-fraction` [*Primary Method*]
(*population* `maximizing-score-mixin`)

⇒ `convergence-fraction` [*Primary Method*]
(*population* `minimizing-score-mixin`)

This function returns the convergence-fraction value which should be used for *population* by the `converged-p` function. The GECO-supplied primary methods for both the `maximizing-score-mixin` and the `minimizing-score-mixin` classes return 0.95. These values are not necessarily the *right* numbers in any real sense, but they are probably reasonable for many applications. Some applications may want to provide different values, and possibly even adaptive methods for specialized subclasses.

⇒ `convergence-threshold-margin` [*Generic Function*]
population

⇒ `convergence-threshold-margin` [*Primary Method*]
(*population* `maximizing-score-mixin`)

⇒ `convergence-threshold-margin` [*Primary Method*]
(*population* `minimizing-score-mixin`)

This function returns the convergence-threshold-margin value which should be used for *population* by the `converged-p` function. The GECO-supplied primary method provided for the `maximizing-score-mixin` class returns 0.95, and the method provided for the `minimizing-score-mixin` class returns 0.05. These values are not necessarily the *right* numbers in any real sense, but they are probably reasonable for many applications. Some applications may want to provide different values, and possibly even adaptive methods for specialized subclasses.

⇒ `as-good-as-test` [*Generic Function*]
population

⇒ `as-good-as-test` [*Primary Method*]
(*population* `maximizing-score-mixin`)

⇒ `as-good-as-test` [*Primary Method*]
(*population* `minimizing-score-mixin`)

This function returns a function of two numeric arguments, which when applied to `scores` from organisms in *population*, indicates whether or not the first score is as good as the second. The GECO-supplied primary method for the `maximizing-score-mixin` class returns `#'>=`, and the method provided for the `minimizing-score-mixin` class returns `#'<=`.

⇒ `better-than-test` [*Generic Function*]
population

⇒ `better-than-test` [*Primary Method*]
(population `maximizing-score-mixin`*)*

⇒ `better-than-test` [*Primary Method*]
(population `minimizing-score-mixin`*)*

This function returns a function of two numeric arguments, which when applied to `scores` from organisms in *population*, indicates whether or not the first score is better than the second. The GECO-supplied primary method provided for the `maximizing-score-mixin` class returns `#'>`, and the method provided for the `minimizing-score-mixin` class returns `#'<`.

⇒ `best-organism` [*Generic Function*]
population

⇒ `best-organism` [*Primary Method*]
(population `maximizing-score-mixin`*)*

⇒ `best-organism` [*Primary Method*]
(population `minimizing-score-mixin`*)*

This function returns the best organism in the corresponding population from population statistics of *population*. The GECO-supplied primary method for the `maximizing-score-mixin` class uses `max-organism`, and the method provided for the `minimizing-score-mixin` class uses `min-organism`.

⇒ `worst-organism` [*Generic Function*]
population

⇒ `worst-organism` [*Primary Method*]
(population `maximizing-score-mixin`*)*

⇒ `worst-organism` [*Primary Method*]
(population `minimizing-score-mixin`*)*

This function returns the best organism in the corresponding population from population statistics of *population*. The GECO-supplied primary method for the `maximizing-score-mixin` class uses `min-organism`, and the method provided for the `minimizing-score-mixin` class uses `max-organism`.

⇒ `best-organism-accessor` *[Generic Function]*
  population

⇒ `best-organism-accessor` *[Primary Method]*
  (population `maximizing-score-mixin`*)*

⇒ `best-organism-accessor` *[Primary Method]*
  (population `minimizing-score-mixin`*)*

This function returns a function which can be applied to an instance of the `population-statistics` class of *population* to obtain the best organism in the corresponding population. The GECO-supplied primary method for the `maximizing-score-mixin` class returns `#'max-organism`, and the method provided for the `minimizing-score-mixin` class returns `#'min-organism`.

⇒ `worst-organism-accessor` *[Generic Function]*
  population

⇒ `worst-organism-accessor` *[Primary Method]*
  (population `maximizing-score-mixin`*)*

⇒ `worst-organism-accessor` *[Primary Method]*
  (population `minimizing-score-mixin`*)*

This function returns a function which can be applied to an instance of the `population-statistics` class of *population* to obtain the worst organism in the corresponding population. The GECO-supplied primary method for the `maximizing-score-mixin` class returns `#'min-organism`, and the method provided for the `minimizing-score-mixin` class returns `#'max-organism`.

3.5 The Organism Class

An **organism** is a member of the population which is being evolved by the GA. Typically an organism represents a single distinct solution to the problem which the GA is set to solve, although sometimes[3] an entire population of organisms cooperate to constitute a solution.

In GECO, an instance of an organism class is a collection of information related to a population member. This may include an explicit representation of the population member (the organism's **phenotype**), or a coded representation (the **genotype**), or both. An evaluation of the organism (its **score**) is also present, so that the GA can have some way to determine which organisms are better than others, and to what extent.

[3]In some kinds of Learning Classifier Systems [HR78, HHNT87], the so-called 'Michigan' approach (for the University of Michigan), each member of a population represents a rule, and the entire population cooperatively evolves as a ruleset. By way of contrast, in the 'Pitt' approach (for the University of Pittsburg) each member of a population represents an entire ruleset.

Typically, during the operation of the GA, the genetic operators manipulate the organism's genotype, and then that is converted into the phenotype, which is then evaluated to produce a score. The genotype typically consists of one or more chromosomes, which encode the features of the phenotype. In some GAs the genotype is bypassed, and the genetic operators manipulate the phenotype directly, in which case the genotype is empty. In other GAs, the organism's score can be determined directly from the genotype, and the conversion from genotype to phenotype is completely omitted. The phenotype is not included in the basic `organism` class, but as a mixin described later (see `organism-phenotype-mixin`, page 39).

⇒ `organism` [*Class*]

Instance Allocated Slots

⇒ `population` [*Slot*]
    `nil`

⇒ `:population` [*Initarg*]

⇒ `population` [*Accessor*]

Provides a link back to the population to which the organism belongs.

⇒ `genotype` [*Slot*]
    `nil`

⇒ `:genotype` [*Initarg*]

⇒ `genotype` [*Accessor*]

A list of zero or more chromosomes, which form an encoded representation of the organism.

⇒ `score` [*Slot*]
    `nil`

⇒ `:score` [*Initarg*]

⇒ `score` [*Accessor*]

A (raw) numeric representation of the value of the organism to the GA, or (initially) `nil`, indicating that the organism hasn't been evaluated.

⇒ `normalized-score` [*Slot*]
    `nil`

⇒ `:normalized-score` [*Initarg*]

⇒ `normalized-score` [*Accessor*]

A normalized version of `score`, with respect to the rest of the population, or `nil`, indicating that the organism either hasn't been evaluated, or that the scores haven't been normalized.

Instance Creation and Initialization

The generic function `make-organism` (see page 22) is the GECO interface for creation of `organism` instances.

The initialization for instances of `organism` has been extended to support the following additional initargs:

⇒ `:random` [*Initarg*]

The initialization for organism instances has also been extended to check the `genotype` slot, and if it is null it will create chromosomes for the organism, using the `make-chromosomes` function, passing the value of the *:random* keyword argument. This is intended to support automatic initialization of the initial population.

⇒ `:no-chromosomes` [*Initarg*]

When non-`nil`, this initarg suppresses creation of the new organism's chromosomes.

Specialized Methods

⇒ `print-object` [*Primary Method*]
    (organism `organism`*) stream*

This method specializes the standard Common Lisp `print-object` function for organisms. It uses the standard Common Lisp function `print-unreadable-object`, includes the type and identity of *organism*, and also causes their `normalized-score` and genotype to be included in the printed representation.

⇒ `copy-organism` [*Generic Function*]
organism `&key` *:new-population*

⇒ `copy-organism` [*Primary Method*]
(organism `organism`*)* `&key` *(:new-population* `(population` *organism*`)`*)*

This function creates and returns a copy of *organism*, modified to be in the population specified by the *:new-population* argument. The `scores` (neither `score` nor `normalized-score`) of *organism* are *not* copied to the new organism (see `copy-organism-with-score`). The GECO-supplied primary method will always return an organism of the same class as *organism*, and uses `copy-chromosome` to copy each chromosome in the genotype of *organism* to initialize the genotype of the returned organism.

This function would generally be used to make a copy which will be modified (*e.g.*, by a genetic operator), thereby invalidating its score.

When using `organism-phenotype-mixin`, it is important to be sure that the `phenotype` slot is copied properly when copying an organism. Depending on the representation of the phenotype, it may or not be worthwhile to copy it whether or not it will subsequently be modified by genetic operators. In any case, copying anything more complex than an atom requires consideration of application and representation specific details.

It may be desirable to define an `:around` method on either `copy-organism` or `copy-organism-with-score` to copy the phenotype (though it should only be necessary to specialize one of these functions, not both). Alternatively, a specialized class's primary method (on one of these functions) could use `call-next-method` to invoke the primary method of class `organism`. If using an `:around` method, don't forget to return the copy.

⇒ `copy-organism-with-score` [*Generic Function*]
organism `&key` *:new-population*

⇒ `copy-organism-with-score` [*Primary Method*]
(organism `organism`*)* `&key` *(:new-population* `(population` *organism*`)`*)*

Creates a copy of the organism in the population specified by the *:new-population* argument, which defaults to the same population as *organism*. The `score` *is* copied to the new organism (see `copy-organism`). The `normalized-score` is not copied on the assumption that the new organism will be part of a new population, and therefore the `normalized-score` will need to be recomputed within the context the rest of the new population. The GECO-supplied primary method uses `copy-organism` to create the new organism.

If *organism* is an instance of a class which includes `organism-phenotype-mixin` as one of its superclasses, refer to the discussion under `copy-organism`, above, regarding copying the `phenotype` slot.

⇒ `make-chromosomes` [*Generic Function*]
organism `&key` *:random*

⇒ `make-chromosomes` [*Primary Method*]
(*organism* `organism`) `&key` *:random*

This function makes and returns a complete set of chromosomes for *organism*. If *:random* is non-`nil`, the chromosomes will have random alleles. The GECO-supplied primary method makes each chromosome with `make-chromosome`, and passes it the *:random* argument. The classes of the chromosomes are obtained by calling the `chromosome-classes` function. The new chromosomes are collected into a list in the same order as the classes returned from `chromosome-classes`, and stored in the `genotype` slot of *organism*, and also returned as the result of the function. This method makes no attempt to determine the proper size for each chromosome, relying on lower level methods to determine this (see page 42).

⇒ `make-chromosome` [*Generic Function*]
organism chromosome-class `&key` *:size :random*

⇒ `make-chromosome` [*Primary Method*]
(*organism* `organism`) *chromosome-class* `&key` *:size :random*

This function provides an abstract interface to creation of an instance of the class *chromosome-class* which will become part of the genotype of *organism*. If *:random* is non-`nil`, the chromosomes will have random alleles. The *:size* argument may be used to control the size of the new chromosome, *i.e.*, the size of its loci vector. The *organism* argument is present so that the chromosome can have a back-link to *organism*, and so that subclasses of *organism* can specialize the chromosome creation process based upon the organism for which the chromosome is intended. The GECO-supplied primary method passes the *:size* and *:random* arguments to `make-instance`.

⇒ `chromosome-classes` [*Generic Function*]
organism

This function returns a list of classes to be used to create chromosomes for instances of the class of of *organism*. The list should contain one class for each chromosome of *organism*, and the order of the classes will determine the order of the chromosomes in the organism instances. The GA developer *must implement the primary method* for all subclasses of the class `organism`. GECO does not provide a default primary method specialized on the `organism` class.[4]

[4]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

⇒ `randomize-chromosomes` [*Generic Function*]
organism

⇒ `randomize-chromosomes` [*Primary Method*]
(*organism* `organism`)

This function replaces all of the chromosomes belonging to *organism* (if any) with randomly chosen chromosomes of the appropriate classes for *organism*. The GECO-supplied primary method determines the appropriate classes for the chromosomes by calling the function `chromosome-classes`, and uses the `pick-random-alleles` function so that the chosen alleles will be valid for each locus of each chromosome.

⇒ `genotype-printable-form` [*Generic Function*]
organism

⇒ `genotype-printable-form` [*Primary Method*]
(*organism* `organism`)

This function returns a single string which is composed of the printable forms of each chromosome belonging to *organism*. The GECO-supplied primary method obtains the printable form of each chromosome using the `~A` format directive with `format`, and concatenates them, with a space between each chromosome's string. The chromosomes' strings are in the same order as the chromosomes in the `genotype` slot of *organism*.

⇒ `evaluate` [*Generic Function*]
thing genetic-plan

⇒ `evaluate` [*:After Method*]
(*organism* `organism`) (*plan* `genetic-plan`)

This function evaluates *organism* and return it's `score`, saving it in *organism*'s `score` slot. Evaluating an organism is generally the most expensive (computationally) operation a GA performs, therefore saving the score to prevent future evaluations of the organisms is almost always worthwhile. For the same reason, it behooves the GA developer to make the evaluation process as efficient as possible.

The GA developer *must implement the primary method* for all subclasses of the class `organism`. GECO does not provide a default primary method specialized on the `organism` class.[5] It is the responsibility of this primary method to perform the calculation of *organism*'s `score`, to store it in *organism*'s `score` slot, and to return it as the result of the function.

The GECO-supplied `:after` method on class `organism` increments the ecosystem's `evaluation-number`.

[5]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

⇒ `normalize-score` [*Generic Function*]
organism population-statistics genetic-plan

⇒ `normalize-score` [*Primary Method*]
(organism `organism`*) (statistics* `population-statistics`*)*
(genetic-plan `genetic-plan`*)*

This function computes the normalized value of *organism*'s `score`, storing the result in the `normalized-score` slot. The invocation of functions responsible for collection of statistics and normalization of scores is handled automatically by GECO. The GECO-supplied primary method uses values from *statistics* to calculate the normalized score as follows:

$$\frac{\texttt{score}_{organism} - \texttt{min-score}_{statistics}}{\texttt{max-score}_{statistics} - \texttt{min-score}_{statistics}}$$

Note that this formula distributes the normalized scores over the interval [0:1]. This results in normalized scores which are (in general) *not* proportional to fitness, since all organisms with the minimum fitness will have normalized scores of zero.

⇒ `eidetic` [*Generic Function*]
thing-1 thing-2

⇒ `eidetic` [*Primary Method*]
(organism-1 `organism`*) (organism-2* `organism`*)*

This function is a predicate, returning true if the organism arguments are equal, *i.e.*, of the same class and have equal chromosomes. The GECO-supplied primary method determines equality of chromosomes by calling the function `eidetic` on each of the chromosomes of the argument organisms.[6]

⇒ `pick-random-chromosome` [*Generic Function*]
organism

⇒ `pick-random-chromosome` [*Primary Method*]
(organism `organism`*)*

This function returns a random chromosome from *organism*. The GECO-supplied primary method uses `pick-random-chromosome-index` to pick the chromosome to return.

[6]I have been questioned regarding the use of the term `eidetic`, above. From Webster's Third New International Dictionary of the English Language, Unabridged: "*eidetic*: of, relating to, or having the characteristics of eide, essences, forms, or images. Further: *eide*: plural of eidos, and *eidos*: something that is seen or intuited: a) in Platonism: idea, b) in Aristotelianism (1): form, essence (2): species." Thus, eidetic can be used to indicate 'of the same species,' which is the essence of my original intent.

⇒ `pick-random-chromosome-index` [*Generic Function*]
organism

⇒ `pick-random-chromosome-index` [*Primary Method*]
(organism `organism`*)*

This function returns a random index into the list of chromosomes belonging to *organism*. The GECO-supplied primary method biases the selection by the relative sizes of each chromosome.

3.5.1 Basic Genetic Operators

⇒ `mutate-organism` [*Generic Function*]
organism `&key` *:chromosome-index :chromosome :locus-index*

⇒ `mutate-organism` [*Primary Method*]
(organism `organism`*)* `&key`
(:chromosome-index `(pick-random-chromosome-index` *organism*`)`*)*
(:chromosome `(nth` *chromosome-index* `(genotype` *organism*`))`*)*
(:locus-index `(pick-locus-index` *chromosome*`)`*)*

This function mutates *organism* randomly. The keyword arguments can be used to control which particular chromosome to mutate, and where it should be mutated. The GECO-supplied primary method mutates the chromosome of *organism* indicated by either the *:chromosome-index* argument or the *:chromosome* argument, picking it randomly otherwise, as shown above. The locus to mutate is specified by the *:locus-index* argument, which is otherwise chosen randomly, as shown above. The actual mutation of the chromosomes is performed by calling the function `mutate-chromosome`.

⇒ `cross-organisms` [*Generic Function*]
parent-1 parent-2 child-1 child-2 `&key` *:chromosome-index :locus-index*

⇒ `cross-organisms` [*Primary Method*]
(parent-1 `organism`*)* *(parent-2* `organism`*)* *(child-1* `organism`*)* *(child-2* `organism`*)*
`&key` *(:chromosome-index* `(pick-random-chromosome-index` *parent-1*`)`*)*
(:locus-index `(pick-locus-index (nth` *chromosome-index* `(genotype` *parent-1*`)))`*)*

This function performs a simple crossover between the two parent organisms *parent-1* and *parent-2*, storing the results in the two child organisms *child-1* and *child-2*. The keyword arguments can be used to control which particular chromosomes to affect and where. The GECO-supplied primary method performs the crossover on the chromosome from both parents indicated by *:chromosome-index* at the locus indicated by *:locus-index*, choosing them randomly otherwise, as shown above. The actual crossover of the chromosomes is performed by calling the function `cross-chromosomes`.

⇒ `uniform-cross-organisms` [*Generic Function*]
parent-1 parent-2 child-1 child-2 `&key` *:chromosome-index*

⇒ `uniform-cross-organisms` [*Primary Method*]
(*parent-1* `organism`) (*parent-2* `organism`) (*child-1* `organism`) (*child-2* `organism`)
`&key` (*:chromosome-index* `(pick-random-chromosome-index` *parent-1*`)`) (*:bias* `0.5`)

This function performs a uniform crossover [Sys89, SD91, Dav91] between the two parent organisms *parent-1* and *parent-2*, storing the result in the two child organisms *child-1* and *child-2*. The keyword arguments can be used to control which particular chromosomes to affect and where. The GECO-supplied primary method performs the crossover on the chromosome from both parents indicated by *:chromosome-index*, choosing it randomly otherwise, as shown above, and using a bias as indicated *:bias* argument, defaulting as shown above if it is not specified. The actual crossover is performed by calling `uniform-cross-chromosomes`.

⇒ `2x-cross-organisms` [*Generic Function*]
parent-1 parent-2 child-1 child-2 `&key` *:chromosome-index :locus-index1*
:locus-index2

⇒ `2x-cross-organisms` [*Primary Method*]
(*parent-1* `organism`) (*parent-2* `organism`) (*child-1* `organism`) (*child-2* `organism`)
`&key` (*:chromosome-index* `(pick-random-chromosome-index` *parent-1*`)`)
(*:locus-index1* `(pick-locus-index (nth` *chromosome-index* `(genotype` *parent-1*`)))`)
(*:locus-index2* `(pick-locus-index (nth` *chromosome-index* `(genotype` *parent-1*`)))`)

This function performs a two-point crossover between the two parent organisms *parent-1* and *parent-2*, storing the result in the two child organisms *child-1* and *child-2*. The keyword arguments can be used to control which particular chromosomes to affect and where. The GECO-supplied primary method performs the crossover on the chromosome from both parents indicated by *:chromosome-index*, choosing it randomly otherwise, as shown above. The actual crossover of chromosomes is performed between the two sites specified by *:locus-index1* and *:locus-index2* (which default as shown above, to randomly chosen sites) by calling the function `2x-cross-chromosomes`.

⇒ `r3-cross-organisms` [*Generic Function*]
parent-1 parent-2 child-1 child-2 `&key` *:chromosome-index :allele-test*

⇒ `r3-cross-organisms` [*Primary Method*]
(parent-1 `organism`*) (parent-2* `organism`*) (child-1* `organism`*) (child-2* `organism`*)*
`&key` *(:chromosome-index* `(pick-random-chromosome-index` *parent-1*`)`*)*
(:allele-test `#'eql`*)*

This function performs a random respectful recombination crossover [Rad92a, Rad92b] between the two parent organisms *parent-1* and *parent-2*, storing the result in the two child organisms *child-1* and *child-2*. The keyword arguments can be used to control which particular chromosomes to affect and where. The GECO-supplied primary method performs the crossover on the chromosome from both parents indicated by *:chromosome-index*, choosing it randomly otherwise, as shown above, and using the *:allele-test* argument to specify a function to tell when two alleles are the same, defaulting as shown above if unspecified. The actual crossover of chromosomes is performed by calling the function `r3-cross-chromosomes`.

⇒ `pmx-cross-organisms` [*Generic Function*]
parent-1 parent-2 child-1 child-2 `&key` *:allele-test :chromosome-index :locus-index1 :locus-index2*

⇒ `pmx-cross-organisms` [*Primary Method*]
(parent-1 `organism`*) (parent-2* `organism`*) (child-1* `organism`*) (child-2* `organism`*)*
`&key` *(:allele-test* `#'eql`*)*
(:chromosome-index `(pick-random-chromosome-index` *parent-1*`)`*)*
(:locus-index1 `(pick-locus-index (nth` *chromosome-index* `(genotype` *parent-1*`)))`*)*
(:locus-index2 `(pick-locus-index (nth` *chromosome-index* `(genotype` *parent-1*`)))`*)*

This function performs a partially mapped crossover (PMX) [Gol89] between the two parent organisms *parent-1* and *parent-2*, storing the result in the two child organisms *child-1* and *child-2*. The keyword arguments can be used to control which particular chromosomes to affect and where. The GECO-supplied primary method performs the crossover on the chromosome from both parents indicated by *:chromosome-index*, which should indicate a `sequence-chromosome`, choosing it randomly otherwise as shown above, and using the *:allele-test* argument to specify a function to tell when two alleles are the same, defaulting as shown above if unspecified. Note that if *:chromosome-index* is not specified, all the chromosomes should be sequence chromosomes, since PMX is only defined for sequence chromosomes, and the chromosome will be chosen randomly. The actual crossover of chromosomes is performed between the two sites specified by *:locus-index1* and *:locus-index2* (which default as shown above, to randomly chosen sites) by calling the function `pmx-cross-chromosomes`.

3.6 Organism Mixin Classes

Presently, there is only one mixin class intended to be used with organism classes.

⇒ `organism-phenotype-mixin` [*Class*]

This class is intended to be mixed with organism classes which need to have a `phenotype` represented for each `organism`. It is an abstract (non-instantiable) class.

Often it is necessary to decode the `genotype` into a phenotype before the organism can be evaluated and assigned a score. Also, some GAs bypass the encoded genotype and use only the phenotype, requiring specially crafted genetic operators which manipulate the phenotype directly.

Note that users of this class should review the discussion regarding copying the `phenotype` slot included in the description of `copy-organism` (page 32).

Instance Allocated Slots

⇒ `phenotype` [*Slot*]
⇒ `:phenotype` [*Initarg*]
⇒ `phenotype` [*Accessor*]

An explicit representation of the organism, *i.e.*, its realization.

Specialized Methods

⇒ `decode` [*Generic Function*]
organism

This function converts *organism*'s `genotype` to it's `phenotype`, and stores it in the `phenotype` slot. GECO automatically invokes `decode`, when appropriate, for instances of `organism-phenotype-mixin` subclasses (see `evaluate`, above).

The GA developer *must implement the primary method* for all subclasses of the class `organism-phenotype-mixin` which performs the decoding operation required by the GA application. GECO does not provide a default primary method specialized on the `organism-phenotype-mixin` class.[7]

⇒ `evaluate` [*Generic Function*]
thing genetic-plan

⇒ `evaluate` [*:Before Method*]
(organism `organism-phenotype-mixin`*) (plan* `genetic-plan`*)*

This function evaluates *organism* and return it's `score`. GECO provides a `:before` method on class `organism-phenotype-mixin`, which invokes the generic function `decode` on *organism*, so that the genotype will be decoded into a phenotype which can be used by the primary method of `evaluate` (see page 34).

[7]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

3.7 The Chromosome Class

An organism's genotype is made up of one or more chromosomes, which contain the encoded genetic representation of what makes the organism different from other organisms. The actual encoding scheme used may vary between different types of organisms, and even between chromosomes of a single type of organism. GECO implements much of the functionality of chromosomes independently of the type of encoding used by the chromosome, but also provides some explicit support for some of the most common kinds of chromosomes via subclasses of the class `chromosome` (see Section 3.8).

⇒ `chromosome` [*Class*]

This class is the basic class upon which all chromosome classes are based. It is an abstract (non-instantiable) class.

Instance Allocated Slots

⇒ `organism` [*Slot*]
  `nil`

⇒ `:organism` [*Initarg*]

⇒ `organism` [*Accessor*]

This slot points back to the organism to which the chromosome belongs.

⇒ `loci` [*Slot*]

⇒ `:loci` [*Initarg*]

⇒ `loci` [*Accessor*]

This slot contains the loci-vector, which is generally a simple, one-dimensional array, whose elements jointly encode the genetic information of the chromosome. Note that the individual loci need not all be of the same type, though they usually are.

Instance Creation and Initialization

The generic function `make-chromosome` (see page 33) is the GECO interface for creation of `chromosome` instances.

The initialization for chromosome instances has been extended to support the following additional initargs:

⇒ `:random` [*Initarg*]

A non-`nil` value for this initarg indicates that each locus should be initialized to a random allele. The value of this keyword is passed to the `make-loci-vector` function, and is intended to support automatic generation of the initial population, and/or creation of random organisms which could be added to a population to increase or restore its diversity.

⇒ `:size` [*Initarg*]

The value of this keyword determines the size of the chromosome, *i.e.*, the size of the loci vector of the chromosome. If its value is `nil`, or it is unspecified, the function `size` is invoked on the new instance. Specialization of the `size` function for the instantiable chromosome class is the normal way to control the size of chromosome instances.

Specialized Methods

⇒ `make-loci-vector` [*Generic Function*]
chromosome size `&key` *:random*

⇒ `make-loci-vector` [*Primary Method*]
(chromosome `chromosome`*) size* `&key &allow-other-keys`

⇒ `make-loci-vector` [*:Around Method*]
(chromosome `chromosome`*) size* `&key` *:random*

This function creates a loci-vector for *chromosome* of size *size* and puts it into the `loci` slot of *chromosome*. The GECO-supplied primary method creates an array whose element-type is `fixnum`, with all the elements initialized to zero (0). The GECO-supplied `:around` method examines the value of the *:random* argument, and if it is non-`nil` passes *chromosome* to `pick-random-alleles`. Since the `:around` method processes the *:random* argument, the primary method uses the `&key &allow-other-keys` sequence to avoid processing it.

⇒ `locus-arity` [*Generic Function*]
    chromosome locus-index

This function returns the number of allele values which are allowed at the locus indicated by *locus-index* in *chromosome*. No primary method is predefined for the general class *chromosome*, but one *must be implemented* for any instantiable chromosome class.[8] Note that locus arity may be a function of *locus-index*, though this is relatively uncommon.

⇒ `copy-chromosome` [*Generic Function*]
    chromosome owner-organism

⇒ `copy-chromosome` [*Primary Method*]
    (*chromosome* `chromosome`) *owner-organism*

This function returns a copy of *chromosome*, setting the `organism` slot of the new chromosome to *owner-organism*. The GECO-supplied primary method makes the copy using `make-chromosome`, passing it the class of *chromosome*, and initializes the loci vector by assigning each locus the same value as the corresponding locus of *chromosome*. Note that this method of copying the alleles may not be appropriate for some chromosome classes, *e.g.*, ones whose loci vectors are not atomic, and which may be manipulated (changed) in ways which might affect more than one organism.

⇒ `print-object` [*Primary Method*]
    (*chromosome* `chromosome`) *stream*

This method specializes the standard Common Lisp `print-object` function for chromosomes. It uses the standard Common Lisp function `print-unreadable-object`, includes the type and identity of *chromosome*, and also uses `loci-printable-form` to include a representation of the alleles of *chromosome*.

⇒ `eidetic` [*Generic Function*]
    thing-1 thing-2

⇒ `eidetic` [*Primary Method*]
    (*chromosome-1* `chromosome`) (*chromosome-2* `chromosome`)

This function is a predicate, returning true (non-`nil`) if the arguments are the same. In the case of instances of chromosome classes, being the same means that they are of the same class, have the same size, and the same alleles at corresponding loci in their loci vectors. The GECO-supplied primary method compares the alleles (which are expected to be allele codes, see Section 3.7.1, page 45) using `#'=`.

[8]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

⇒ `size` [*Generic Function*]
  thing

⇒ `size` [*Primary Method*]
  (chromosome `chromosome`*)*

This function returns the size of its argument in whatever units are appropriate. The GECO-supplied primary method for `chromosome` returns the size of the loci vector belonging to *chromosome*.

⇒ `pick-locus-index` [*Generic Function*]
  chromosome

⇒ `pick-locus-index` [*Primary Method*]
  (chromosome `chromosome`*)*

This function returns a random index into the loci vector of *chromosome*. The GECO-supplied primary method calls `geco-random-integer` with the size of *chromosome*.

⇒ `hamming-distance` *[Generic Function]*
chromosome-1 chromosome-2

⇒ `hamming-distance` *[Primary Method]*
(chromosome-1 `chromosome`*) (chromosome-2* `chromosome`*)*

This function returns the count of the number of loci in the two arguments which have different alleles at corresponding loci. The GECO-supplied primary method compares the number of loci which are in *chromosome-1*, and uses `#'=` to compare the allele codes (see Section 3.7.1, below). It is an error if the entire part of *chromosome-2* designated is not within its loci vector, *i.e.*, if an invalid locus index is implied by the arguments.

3.7.1 Allele Coding: Codes vs. Values

Fixnums are chosen as the default type for loci-vector elements because they can frequently be stored more efficiently than general lisp values, particularly when there are only a small number of alleles per locus. To make this choice more generally useful, GECO interprets the values stored in loci-vector elements as allele codes, as opposed to allele values. This allows a straightforward conversion between these fixnums and the actual alleles via simple table lookups, *i.e.*, the table contains the allele values and is indexed by the allele code. GECO supports this translation directly via the generic functions `allele-values` and `allele-code-to-value`. GECO also supports conversion of the allele codes to printable form via the generic functions `printable-allele-values`, `loci-printable-form` and `locus-printable-form`. These functions are describe below. Note that the descriptions of some functions may gloss over the distinction between allele codes and allele values, referring to either of them simply as alleles, but it should be clear from context which is being manipulated.

⇒ `pick-random-alleles` *[Generic Function]*
chromosome

⇒ `pick-random-alleles` *[Primary Method]*
(chromosome `chromosome`*)*

This function initializes the loci of *chromosome*'s `loci-vector` to random alleles. The GECO-supplied primary method calls `pick-random-allele` for each locus to obtain its new allele.

⇒ `pick-random-allele` [*Generic Function*]
chromosome locus-index

⇒ `pick-random-allele` [*Primary Method*]
(*chromosome* `chromosome`) *locus-index*

This function returns a random allele code for the indicated locus of *chromosome*. The GECO-supplied primary method selects a random number in the proper range by calling `geco-random-integer` with the value returned by the `locus-arity` function for the indicated chromosome and locus-index.

⇒ `allele-code-to-value` [*Generic Function*]
chromosome locus-index allele-code

⇒ `allele-code-to-value` [*Primary Method*]
(*chromosome* `chromosome`) *locus-index allele-code*

This function converts *allele-code* to an allele value (see page 45). The *chromosome* and *locus-index* arguments permit different loci of different chromosomes to have different mappings (codings) between allele codes and allele values. In particular, this permits different chromosomes/loci to have different arity. The GECO-supplied primary method uses `aref` to index into the array returned by `allele-values`.

⇒ `allele-values` [*Generic Function*]
chromosome locus-index

This function returns a vector of allele values, which may be used to convert the allele codes used in loci vectors. GECO *does not* implement a primary method for this function for the `chromosome` class. Instantiable chromosome classes should implement this method based on the genetic representation they use.[9]

Note that it is generally preferable to use the function `allele-code-to-value`, rather than indexing into the vector returned by this function, since the implementation may permit a more efficient implementation than is supported by this general mechanism (*e.g.*, for subclasses of `binary-chromosome`).

[9]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

⇒ `printable-allele-values` [*Generic Function*]
chromosome locus-index

This function returns a vector of characters indexed by **allele code** to generate a printable representation for a chromosome. GECO *does not* implement a primary method for this function for the `chromosome` class. Instantiable chromosome classes should implement this method based on the genetic representation they use.[10]

Note that it is generally preferable to call one of the functions `loci-printable-form` or `locus-printable-form`, rather than indexing into the vector returned by this function, since their implementation may permit a more efficient implementation than is supported by this general mechanism (*e.g.*, for subclasses of `binary-chromosome`).

⇒ `loci-printable-form` [*Generic Function*]
chromosome

⇒ `loci-printable-form` [*Primary Method*]
(chromosome `chromosome`*)*

This function returns a string which is a printable representation of the `loci-vector` of *chromosome*. The GECO-supplied primary method constructs a string whose length is the size of *chromosome*, and whose characters represent the alleles of the chromosome on a one-for-one basis, with the first character corresponding to the first locus' allele, *etc.* The characters representing each locus' allele are determined by calling `locus-printable-form`.

⇒ `locus-printable-form` [*Generic Function*]
chromosome locus-index

⇒ `locus-printable-form` [*Primary Method*]
(chromosome `chromosome`*) locus-index*

This function returns the character which represents the allele at *locus-index* in *chromosome*. The GECO-supplied primary method uses `aref` to index into the vector returned by `printable-allele-values` with the **allele code** found at *locus-index* in *chromosome*. If the allele code is not a valid index for `printable-allele-values`, return `#\?`.

⇒ `locus` [*Generic Function*]
chromosome locus-index

⇒ `locus` [*Primary Method*]
(chromosome `chromosome`*) locus-index*

This function returns the **allele code** at *locus-index* in the `loci` of *chromosome*.

[10]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

⇒ `(setf locus)` [*Generic Function*]
allele-code chromosome locus-index

⇒ `(setf locus)` [*Primary Method*]
allele-code (chromosome `chromosome`*) locus-index*

This function stores the *allele-code* into the locus indicated by *locus-index* in the loci vector of *chromosome*.

Note that Common Lisp has rather non-intuitive ordering for the arguments for `setf` functions and methods. An example of proper invocation is:

```
(setf (locus chromosome locus#) allele-code)
```

⇒ `count-allele-codes` [*Generic Function*]
chromosome from-index loci-to-count allele-code

⇒ `count-allele-codes` [*Primary Method*]
(chromosome `chromosome`*) from-index loci-to-count allele-code*

This function returns the count of the number of loci in part of the *chromosome* which have *allele-code* in them. The part of *chromosome* in which the count is conducted are specified as starting at the locus whose index is *from-index* and which is *loci-to-count* long. The GECO-supplied primary method compares the alleles to *allele-code* using `#'=`. It is an error if the entire part of *chromosome* designated is not within the loci vector, *i.e.*, if an invalid locus index is implied by the arguments *from-index* and *loci-to-count*.

3.7.2 Basic Chromosomal Genetic Operators

⇒ `mutate-chromosome` [*Generic Function*]
chromosome locus-index

⇒ `mutate-chromosome` [*Primary Method*]
(chromosome `chromosome`*) locus-index*

This function mutates *chromosome* at the locus *locus-index*. The GECO-supplied primary method uses `pick-random-allele` to choose the new allele code for the locus.

Note that for instances of subclasses of `binary-chromosome`, this implementation will produce on average one mutation for every two invocations of this function, since half the time the randomly chosen allele will be the same as the current allele at the indicated locus.

⇒ `cross-chromosomes` [*Generic Function*]
parent-1 parent-2 child-1 child-2 locus-index

⇒ `cross-chromosomes` [*Primary Method*]
(parent-1 `chromosome`*) (parent-2* `chromosome`*) (child-1* `chromosome`*)*
(child-2 `chromosome`*) locus-index*

This function performs a simple crossover operation between the two parent chromosomes, storing the results in the two child chromosomes, using *locus-index* as a control parameter for the crossover. The GECO-supplied primary method performs a conventional one-point crossover, assumes all the chromosomes are the same size and of compatible classes, the *child-1* receives *locus-index* alleles from *parent-1*, and the remaining alleles from *parent-2*; *child-2* gets its alleles in an analogous manner.

⇒ `uniform-cross-chromosomes` [*Generic Function*]
parent-1 parent-2 child-1 child-2 `&key` *:bias*

⇒ `uniform-cross-chromosomes` [*Primary Method*]
(parent-1 `chromosome`*) (parent-2* `chromosome`*) (child-1* `chromosome`*)*
(child-2 `chromosome`*)* `&key` *(:bias* `0.5`*)*

This function performs a uniform crossover [Sys89, SD91, Dav91] operation between the two parent chromosomes, storing the results in the two child chromosomes, using the *:bias* argument as a control parameter for the crossover. The GECO-supplied primary method performs a conventional uniform crossover, assumes all the chromosomes are the same size and of compatible classes, *child-1* statistically receives a fraction of the alleles specified by *:bias* from *parent-1*, and the remaining alleles from *parent-2*; *child-2* gets its alleles in an analogous manner.

⇒ `2x-cross-chromosomes` [*Generic Function*]
parent-1 parent-2 child-1 child-2 locus-index1 locus-index2

⇒ `2x-cross-chromosomes` [*Primary Method*]
(parent-1 `chromosome`*) (parent-2* `chromosome`*) (child-1* `chromosome`*)*
(child-2 `chromosome`*) locus-index1 locus-index2*

This function performs a two-point crossover operation between the two parent chromosomes, storing the results in the two child chromosomes. The GECO-supplied primary method performs a conventional two-point crossover, assumes all the chromosomes are the same size and of compatible classes. Alleles between *locus-1* and *locus-2* are copied from from *parent-1* to *child-2*, and the remaining alleles from *parent-2*; *child-1* gets its alleles in an analogous manner. If *locus-1* is greater than *locus-2*, the copy operation wraps around from the end of the chromosome back to its beginning, then copies from the beginning to *locus-2*.

⇒ `swap-alleles` [*Generic Function*]
  chromosome `&key` *:locus-index :locus-index2*

⇒ `swap-alleles` [*Primary Method*]
  (chromosome `chromosome`*)* `&key` *(:locus-index* `(pick-locus-index` *chromosome*`)`*)*
  (:locus-index2 `(mod (1+` *locus-index*`) (size` *chromosome*`))`*)*

This function swaps alleles between two loci of *chromosome*. The two loci to swap are indicated by the arguments *:locus-index* and *:locus-index2*. The GECO-supplied primary method allows the keyword arguments to default as shown above.

⇒ `scramble-alleles` [*Generic Function*]
  chromosome

⇒ `scramble-alleles` [*Primary Method*]
  (chromosome `chromosome`*)*

This function randomly rearranges the alleles of *chromosome*. The *chromosome* will have the same set of **allele codes** both before and after the operation, but they will appear in a different permutation on the loci. Note that this operator should not be applied to chromosomes for which the arity of all loci is not the same.

3.8 Subclasses of Chromosome

GECO provides some support for some of the more common kinds of chromosomes. Presently, this includes:

- Binary chromosomes
- Sequence chromosomes

This section also describes some support provided for decoding binary coded chromosomes.

3.8.1 Binary Chromosomes

⇒ `binary-chromosome` [*Class*]
chromosome

Binary chromosomes are a subclass of `chromosome` whose alleles are always chosen from the set $\{0\ 1\}$. This restriction allows them to be represented more efficiently, and specialized methods can be provided which process them somewhat more efficiently than the more general case.

Note that `binary-chromosome` is still an abstract (non-instantiable) class, since the size of the chromosome is left unspecified.

This class has no additional slots beyond those defined for the `chromosome` class.

Instance Creation and Initialization

The generic function `make-chromosome` (see page 33) is the GECO interface for creation of `chromosome` instances.

Specialized Methods

⇒ `locus-arity` [*Generic Function*]
chromosome locus-index

⇒ `locus-arity` [*Primary Method*]
(chromosome `binary-chromosome`*) locus-index*

This function returns the number of allele values which are allowed at the locus indicated by *locus-index* in *chromosome*. The GECO-supplied primary method always returns 2, regardless of the value of *locus-index*.

⇒ `allele-code-to-value` [*Generic Function*]
chromosome locus-index allele-index

⇒ `allele-code-to-value` [*Primary Method*]
(chromosome `binary-chromosome`*) locus-index allele-index*

This function converts *allele-code* to an allele value (see the discussion on allele coding in Section 3.7.1). The GECO-supplied primary method simply returns the allele code, since the value and the code are the same.

⇒ `allele-values` [*Generic Function*]
chromosome locus-index

⇒ `allele-values` [*Primary Method*]
(*chromosome* `binary-chromosome`) *locus-index*

This function returns a vector of allele values, which may be used to convert the allele codes used in loci-vectors. The GECO-supplied primary method for `binary-chromosome` always returns the vector `#(0 1)`, regardless of the value of *locus-index*.

⇒ `printable-allele-values` [*Generic Function*]
chromosome locus-index

⇒ `printable-allele-values` [*Primary Method*]
(*chromosome* `binary-chromosome`) *locus-index*

This function returns a vector of characters which may be indexed by allele code to generate a printable representation of *chromosome*. The GECO-supplied primary method for `binary-chromosome` always returns the vector `#(#\0 #\1)`, regardless of the value of *locus-index*.

⇒ `make-loci-vector` [*Generic Function*]
chromosome size `&key` *:random*

⇒ `make-loci-vector` [*Primary Method*]
(*chromosome* `binary-chromosome`) *size* `&key &allow-other-keys`

This function creates a loci vector for *chromosome* of size *size* and puts it into the `loci` slot of *chromosome*. The GECO-supplied primary method creates an array whose element-type is `bit`, and with all the elements initialized to zero (0). Since the inherited `:around` method (page 42) processes the *:random* argument, the primary method uses the `&key &allow-other-keys` sequence to avoid processing it.

3.8.2 Binary Chromosome Decoding

⇒ `decode-binary-loci-value` [*Generic Function*]
chromosome from-index loci-to-decode

⇒ `decode-binary-loci-value` [*Primary Method*]
(*chromosome* `binary-chromosome`) *from-index loci-to-decode*

This function returns the numeric value encoded by the loci of *chromosome* which start at the locus indexed by *from-index* and are *loci-to-decode* in length. The GECO-supplied primary method treats the loci as an unsigned binary coded bit string, with the most significant bits having the lower indices in the loci vector.

3.8.3 Gray Code Translation

Sometimes it is advantageous to treat a binary coded value as if it were encoded using a gray code scheme[CS88]. GECO provides a special class whose instances can be used for quickly decoding (or encoding) gray coded binary values.

The conversion scheme implemented by GECO is based on an implementation in `C` by Larry Yaeger `<larryy@apple.com>`, which was published in the GA-List Digest v6n5 (`GA-List@AIC.NRL.Navy.Mil`).

⇒ `gray-code-translation` [*Class*]

A class whose instances support translation between standard binary and gray coded integer values for a specified number of bits.

Instance Allocated Slots

⇒ `number-of-bits` [*Slot*]
⇒ `:number-of-bits` [*Initarg*]
⇒ `number-of-bits` [*Accessor*]

This slot specifies the number of bits in the bit string which will be encoded or decoded. This initarg should be specified when an instance of `gray-code-translation` is created for proper initialization of the instance.

⇒ `b2g-map` [*Slot*]
⇒ `b2g-map` [*Accessor*]

⇒ `g2b-map` [*Slot*]
⇒ `g2b-map` [*Accessor*]

When the `:number-of-bits` initarg is specified at instance creation time, these two slots will be initialized to bit maps which are used by the conversion methods described below.

Instance Creation and Initialization

No special functions for the creation of `ecosystem` instances have been defined, since `make-instance` and the standard CLOS protocol it follows provide all the necessary functionality.

Note that the `:number-of-bits` initarg should be specified when an instance of `gray-code-translation` is created for proper initialization of the instance.

Specialized Methods

⇒ `gray2bin` [*Generic Function*]
translation-instance gray-coded-value

⇒ `gray2bin` [*Primary Method*]
(translation-instance `gray-code-translation`*) value*

This function uses *translation-instance* to convert the gray coded *value* to its binary coded equivalent.

⇒ `bin2gray` [*Generic Function*]
translation-instance gray-coded-value

⇒ `bin2gray` [*Primary Method*]
(translation-instance `gray-code-translation`*) value*

This function uses *translation-instance* to convert the binary coded *value* to its gray coded equivalent.

The following example illustrates the use of these functions.[11]

```
(let ((gct (make-instance 'gray-code-translation
             :number-of-bits 5)))
  (format t "~&Int ~7TBinary ~19TGray ~23TGrayInt  RecoveredInt")
  (dotimes (i (expt 2 (number-of-bits gct)))
    (let ((g (bin2gray gct i)))
      (format t "~%~3D  ~8B  ~8B  ~4D  ~8D"
              i i g g (gray2bin gct g))))
  (format t "~2%GrayInt Int")
  (dotimes (i (expt 2 (number-of-bits gct)))
    (format t "~% ~6D ~3D" i (gray2bin gct i))))
```

[11]The code for this example is included in a comment in the `chromosome-methods.lisp` file.

3.8.4 Sequence Chromosomes

⇒ `sequence-chromosome` [*Class*]
chromosome

Sequence chromosomes are a subclass of `chromosome` whose alleles are always chosen such that every locus of a chromosome has an allele which does not occur at any other locus of the chromosome. This requires that several operations which manipulate these chromosomes be handled differently in order to maintain this property of uniqueness of alleles within the chromosome.

Note that `sequence-chromosome` is still an abstract (non-instantiable) class, since the size of the chromosome and the number of alleles (usually, but not necessarily the same) are left unspecified.

This class has no additional slots beyond those defined for the `chromosome` class.

Instance Creation and Initialization

The generic function `make-chromosome` (see page 33) is the GECO interface for creation of `chromosome` instances.

Specialized Methods

⇒ `pick-random-alleles` [*Generic Function*]
chromosome

⇒ `pick-random-alleles` [*Primary Method*]
(chromosome `sequence-chromosome`*)*

This function initializes the loci of *chromosome* to random alleles. The GECO-supplied primary method assigns allele codes to each locus in *chromosome* corresponding to the locus' index into the loci vector, and the calls `scramble-alleles` on *chromosome*.

3.8.5 Sequence Genetic Operators

⇒ `pmx-cross-chromosomes` [*Generic Function*]

parent-1 parent-2 child-1 child-2 `&key` *:allele-test :locus-index1 :locus-index2*

⇒ `pmx-cross-chromosomes` [*Primary Method*]

(parent-1 `sequence-chromosome`*) (parent-2* `sequence-chromosome`*)*
(child-1 `sequence-chromosome`*) (child-2* `sequence-chromosome`*)* `&key`
(:allele-test `#'eql`*) (:locus-index1* `(pick-locus-index` *parent-1*`)`*)*
(:locus-index2 `(pick-locus-index` *parent-1*`)`*)*

This function performs a partially mapped crossover [Gol89] between the two parent chromosomes *parent-1* and *parent-2*, storing the result in the two child chromosomes *child-1* and *child-2*. The two arguments *:locus-index1* and *locus-index2* specify the boundaries of the segment of *parent-1* which is to be crossed with *parent-2*, defaulting as shown above. The *:allele-test* argument specifies a predicate to determine equality of two alleles, defaulting as shown above. The GECO-supplied primary method treats the chromosome as circular when *:locus-index1* > *:locus-index2*. If *:locus-index1* = *:locus-index2*, or if one is 0 and the other = the length of the parent chromosomes, then the children are simply copies of the parents.

⇒ `r3-cross-chromosomes` [*Generic Function*]

parent-1 parent-2 child-1 child-2 `&key` *:allele-test*

⇒ `r3-cross-chromosomes` [*Primary Method*]

(parent-1 `sequence-chromosome`*) (parent-2* `sequence-chromosome`*)*
(child-1 `sequence-chromosome`*) (child-2* `sequence-chromosome`*)* `&key`
(:allele-test `#'eql`*)*

This function performs a random respectful recombination crossover [Rad92a, Rad92b]) between the two parent chromosomes *parent-1* and *parent-2*, storing the result in the two child chromosomes *child-1* and *child-2*. The *:allele-test* argument specifies a predicate to determine equality of two alleles, defaulting as shown above.

3.9 The Genetic Plan Class

A genetic plan controls the overall strategy which determines how an ecosystem `regenerate`, *i.e.*, how new organisms are created from older organisms. This generally includes the overall scheme for selection of organisms for reproduction and application of genetic operators. The actual selection methods provided by GECO are described in Section 3.10, since they are typically not specialized on the class of the genetic plan.

⇒ `genetic-plan` [*Class*]

Instance Allocated Slots

⇒ `ecosystem` [*Slot*]
⇒ `:ecosystem` [*Initarg*]
⇒ `ecosystem` [*Accessor*]

This slot records the `ecosystem` which is using the genetic plan.

⇒ `generation-limit` [*Slot*]
`nil`
⇒ `:generation-limit` [*Initarg*]
⇒ `generation-limit` [*Accessor*]

⇒ `evaluation-limit` [*Slot*]
`nil`
⇒ `:evaluation-limit` [*Initarg*]
⇒ `evaluation-limit` [*Accessor*]

These slots (which default to `nil`) can be used to establish termination criteria for the evolutionary process. They are used by the GECO-supplied primary method for `evolution-termination-p` (see below).

Instance Creation and Initialization

The generic function `make-genetic-plan` (see page 19) is the GECO interface for creation of `genetic-plan` instances.

Specialized Methods

⇒ `regenerate` *[Generic Function]*
plan thing

⇒ `regenerate` *[Primary Method]*
(plan `genetic-plan`*) (ecosystem* `ecosystem`*)*

⇒ `regenerate` *[Primary Method]*
(plan `genetic-plan`*) (old-population* `generational-population`*)*

This function creates a new version of *thing* which is more evolved according to the genetic plan *plan*. The GECO-supplied version of `regenerate` which is specialized to the class `ecosystem` invokes `regenerate` on *ecosystem*'s population, and saves the result in *ecosystem*'s `population` slot.

Note that `generational-population` is currently the only population class for which `regenerate` is defined.

The GECO-supplied version of `regenerate` which is specialized to the class `generational-population` is not intended to be used for real GAs, but to serve as a template to illustrate the responsibilities of `regenerate`. Therefore a specialized method should be implemented for all subclasses of `population`, including `generational-population`.[12] For generational GAs, the responsibilities of `regenerate` include:

[12]There are comments at the beginning of the `generics.lisp` file which summarize the functions which should or must be defined to implement a working GA using GECO.

- Create a new population of the same class as *old-population*, and whose size is *based on* the size of *old-population*. Note that the new population need not necessarily be the same size as *old-population* unless that is consistent with the genetic plan. Note also that this size is the size of the `organisms` vector, but this vector does not contain any organisms.
- Assure that the `ecosystem` slot of the new population is the same as that of *old-population*.
- Install organisms in the new population, based on the organisms of *old-population*. This typically involves:
  - Selecting some of the organisms from *old-population* to participate in creation of the new population. This selection process is typically based on their `score`s (fitness or penalty), and may be performed using one or more selection methods (see Section 3.10), or similar methods.
  - Copying some of the selected organisms from *old-population*, and
  - Creating new organisms to include in the new population, typically by either mutating selected organisms from *old-population* or combining some of them using other genetic operators such as crossover.

⇒ `evolution-termination-p` [*Generic Function*]
plan

⇒ `evolution-termination-p` [*Generic Function*]
(*plan* `genetic-plan`)

This function is a predicate used by the GECO-supplied method `evolve` to determine when to terminate the evolutionary process. The GECO-supplied primary method returns true (non-`nil`) when either an evaluation limit or a generation limit has been established (by putting a number in the `evaluation-limit` or the `generation-limit` slot of *plan*) and either of those limits has been exceeded, or when `converged-p` (page 25) returns true.

3.10 Selection Methods

GECO provides a sampling of selection methods. None of them are guaranteed to be the best in the world, but some of them may prove useful as examples, or as a base upon which to build your own.

⇒ `pick-random-organism-index` [*Generic Function*]
population

⇒ `pick-random-organism-index` [*Primary Method*]
(population `population`*)*

This function returns the index of a random organism from *population*. The GECO-supplied primary method simply calls `geco-random-integer` with the argument `(size` *population*`)`.

⇒ `pick-random-organism` [*Generic Function*]
population

⇒ `pick-random-organism` [*Primary Method*]
(population `population`*)*

This function returns a random organism from *population*. The GECO-supplied primary method returns the organism from *population* indexed by the value returned from `pick-random-organism-index`.

⇒ `roulette-pick-random-weight-index` [*Function*]
weights-table `&key` *:invert-p*

This function selects a random index into an array of weights *weights-table*, using the roulette wheel approach [Gol89]. An entry in *weights-table* indicates the probability that the corresponding index should be returned. The *:invert-p* argument when non-`nil` causes the selection to be inversely proportional to *weights-table* entries. The GECO-supplied primary method assumes that *weights-table* has been normalized to sum to 1.0.

⇒ `roulette-pick-random-organism-index` [*Primary Method*]
population

⇒ `roulette-pick-random-organism-index` [*Generic Function*]
(population `population`*)*

This function selects a random organism from *population*, weighted by `score`, using the roulette wheel approach [Gol89], as used in DeJong's R1 [DeJ75]; it is also referred to by Brindle as stochastic sampling with replacement [Bri81].

⇒ `roulette-pick-random-organism` *[Primary Method]*
  population

⇒ `roulette-pick-random-organism` *[Generic Function]*
  (population `population`*)*

This function selects a random organism from *population*, weighted by `score`, using the roulette wheel approach [Gol89], as used in DeJong's R1 [DeJ75]; also referred to by Brindle as stochastic sampling with replacement [Bri81].

⇒ `stochastic-remainder-preselect` *[Primary Method]*
  population `&key` *:multiplier*

⇒ `stochastic-remainder-preselect` *[Generic Function]*
  (population `population`*)* `&key` *(:multiplier* `1`*)*

This function prepares and returns a function (actually a closure) of no arguments which will select and return random organisms from *population*, weighted by `score`, using a technique referred to by Brindle as stochastic remainder selection without replacement [Bri81]. Each call to the returned function will return an organism member of *population* until the appropriate number of organisms have been selected, then the function will return `nil`. The *:multiplier* keyword argument can be supplied to indicate the number of organisms to be selected, in terms of the size of *population*. For instance, if it is desired that the returned function return twice as many organisms as are in *population*, a *:multiplier* value of 2 should be used.

The following code fragment illustrates the intended use:

```
(let ((selector (stochastic-remainder-preselect some-population)))
  (do ((organism (funcall selector) (funcall selector)))
      ((null organism))
    (do-something-with organism)))
```

⇒ `ranking-preselect` [*Primary Method*]
population `&key` *:multiplier :max*

⇒ `ranking-preselect` [*Generic Function*]
(*population* `population`) `&key` (*:multiplier* `1`) (*:max* `2.0`)

This function prepares and returns a function (actually a closure) of no arguments which will select and return random organisms from *population*, weighted by the rank of each organism's `score` wthin *population*, without replacement. [Bak85] Each call to the function returned from this method will return an organism member of *population* until the appropriate number of organisms have been selected, then the function will return `nil`. The *:multiplier* keyword argument can be supplied to indicate the number of organisms to be selected, in terms of the size of *population*. For instance, if it is desired that the returned function return twice as many organisms as are in *population*, a *:multiplier* value of 2 should be used.

The main idea of rank selection (as implemented here) is as follows: Sort the population by score from best to worst, assigning a linearly decreasing number of copies to each organism, starting with *:max* copies of the most fit organism. The number of copies of the least fit organism is determined according to the following formula:

$$:max - 2.0(:max - 1.0)$$

where fractional remainders are used as probabilities, and negative values are equivalent to zero. Note that values for *:max* greater than 2.0 will result in some fraction of the less fit organisms in *population* not being selected at all.

⇒ `pick-some-random-organism-indices` [*Generic Function*]
population number-to-pick

⇒ `pick-some-random-organism-indices` [*Primary Method*]
(*population* `population`) *number-to-pick*

This function returns *number-to-pick* random organism indices for *population*. The indices will each be unique, *i.e.*, there will be no duplicates for any given call to this function.

⇒ `tournament-select-organism` [*Generic Function*]
population tournament-size

⇒ `tournament-select-organism` [*Primary Method*]
(*population* `population`) *tournament-size*

This function picks *tournament-size* organisms from *population* at random, and returns the best (most fit) one. The GECO-supplied method calls `pick-some-random-organism-indices` to establish the members of the tournament, and uses `better-than-test` to compare the organisms.

3.11 The Population Statistics Class

This class supports accumulation of information about (at least) the scores of the members of a population. This information can be used for normalizing the scores across the population, *etc.*

Instances are created automatically by GECO at the end of evaluating a new population, after all the organisms have been created and evaluated.

⇒ `population-statistics` [*Class*]

Instance Allocated Slots

⇒ `population` [*Slot*]
⇒ `:population` [*Initarg*]
⇒ `population` [*Accessor*]

This slot indicates the population to which this population-statistics instance applies. The `:population` initarg should be specified when a `population-statistics` instance is created.

⇒ `sum-score` [*Slot*]
⇒ `:sum-score` [*Initarg*]
⇒ `sum-score` [*Accessor*]

⇒ `avg-score` [*Slot*]
⇒ `:avg-score` [*Initarg*]
⇒ `avg-score` [*Accessor*]

⇒ `max-score` [*Slot*]
⇒ `:max-score` [*Initarg*]
⇒ `max-score` [*Accessor*]

⇒ `min-score` [*Slot*]
⇒ `:min-score` [*Initarg*]
⇒ `min-score` [*Accessor*]

⇒ `max-organism` [*Slot*]
⇒ `:max-organism` [*Initarg*]
⇒ `max-organism` [*Accessor*]

⇒ `min-organism` [*Slot*]
⇒ `:min-organism` [*Initarg*]
⇒ `min-organism` [*Accessor*]

These slots hold the calculated values, respectively, for:

- the sum of the `score`s of all the organisms in the `population`
- the average (statistical mean) of the `score`s of all the organisms in the `population`
- the maximum of the `score`s of all the organisms in the `population`
- the minimum of the `score`s of all the organisms in the `population`
- an organism in `population` which had a score of `max-score`
- an organism in `population` which had a score of `min-score`

The above values are calculated by `compute-statistics`, which is invoked automatically at the end of initialization of an instance of a `population-statistics` class.

⇒ `sum-normalized-score` [*Slot*]
⇒ `:sum-normalized-score` [*Initarg*]
⇒ `sum-normalized-score` [*Accessor*]

⇒ `avg-normalized-score` [*Slot*]
⇒ `:avg-normalized-score` [*Initarg*]
⇒ `avg-normalized-score` [*Accessor*]

These slots hold the calculated values, respectively, for:

- the sum of the normalized `score`s of all the organisms in the `population`
- the average (statistical mean) of the normalized `score`s of all the organisms in the `population`

The above values are calculated by `compute-normalized-statistics`, which GECO invokes automatically as part of evaluatiing a population (see figure 2.2, page 14).

Instance Creation and Initialization

The generic function `make-population-statistics` (see page 23) is the GECO interface for creation of `population-statistics` instances.

The instance initialization for this class has been extended to automatically call `compute-statistics` (see below) on the new instance.

⇒ `print-object` [*Primary Method*]
(*self* `population-statistics`) *stream*

This method specializes the standard Common Lisp `print-object` generic function for instances of the `population-statistics` class. It uses the standard Common Lisp function `print-unreadable-object`, includes the type and identity of *self*, and also includes one of the following:

- If the population is converged, the `avg-score` of *self*, which is the value to which all the organisms have converged, else
- Both the `avg-score` and `avg-normalized-score` of *self*.

⇒ `compute-statistics` [*Generic Function*]
  population-statistics

⇒ `compute-statistics` [*Primary Method*]
  (*population-statistics* `population-statistics`)

This function calculates and stores whatever statistics of the population are necessary for the genetic plan to calculate normalized scores of the organisms of the population indicated by the `population` slot. The function is called by a GECO-supplied initialization method on the `population-statistics` class. The GECO-supplied primary method calculates the sum of all the scores of the organisms in the `population`, and the minimum, maximum, and average scores for the `population`, and retains (pointers to) organisms in `population` which have the minimum and maximum scores. These values are stored in the appropriate slots of *population-statistics*.

⇒ `compute-normalized-statistics` [*Generic Function*]
  population-statistics

⇒ `compute-normalized-statistics` [*Primary Method*]
  (*population-statistics* `population-statistics`)

This function calculates and stores whatever statistics of the normalized scores of the `population` are necessary for the genetic plan to control the evolution of the ecosystem at the current time. The function is automatically called by a GECO-supplied `normalize-score` method which is specialized to the `population`, `population-statistics`, and `genetic-plan` classes. The GECO-supplied primary method calculates the sum of all the normalized scores of the organisms in the `population`, and the average normalized score for the `population`. These values are stored in the appropriate slots of *population-statistics*.

Chapter 4

A Simple Binary Example

An example of how to customize GECO is provided in the file `sb-test.lisp`, which presents two alternative GAs to solve a simple problem often called the *count ones* or *onemax* [Ack87], which tries to maximize the number of one-bits in a binary chromosome. The following material provides a overview of the definitions in this file which implement the first example GA, discussing each one, why it is necessary and/or what it does, and how it fits into the GECO framework.

4.1 Using GECO with Packages

The GECO.system file defines the geco package, which contains all the GECO definitions. Normally, a GA application will be defined in its own package or packages. All the examples provided with GECO are defined in the geco-user package, which is also defined in GECO. system, as follows:

```
(defpackage GECO-USER
    (:use "COMMON-LISP"
          #+:ccl-2 "CCL"
          "GECO")
    (:nicknames "GU"))
```

Then, near the beginning of each file containing code in this package, a line should appear which tells lisp that the following code is in the appropriate package, so that it has access to all the GECO definitions. *e.g.*,

```
(in-package :GECO-USER)
```

4.2 Defining the Genetic Structures

First, let's define the class of chromosome we'll need. The most common chromosomes used by GAs are typically bit vectors, *i.e.*, each locus on the chromosome has a binary value. GECO has a predefined subclass of chromosome for just this purpose, `binary-chromosome`, which though it doesn't have any additional slots, does have some specialized methods which support displaying binary chromosomes, and decoding values encoded in them. But `binary-chromosome` is still too general for instantiation, so we define the class `binary-chromosome-10` to add a method `size` which returns 10, the number of bits in the chromosome. Now when GECO instantiates a chromosome of this class, it can determine the size of the chromosome's loci vector by simply using the standard protocol to inquire from the chromosome instance it's size. This allows GECO to allocate the loci vector automatically as part of chromosome instantiation.

```
(defclass BINARY-CHROMOSOME-10 (binary-chromosome)
  ()
  (:documentation
   "A 10-bit binary chromosome."))

(defmethod SIZE ((self binary-chromosome-10))
  "So GECO will know how large to make the chromosome."
  10)
```

Next, we define `simple-binary-10-organism` as a subclass of `organism`. This class will hold a single chromosome of the class we just defined in its `genotype` slot. Using the same technique as in the previous paragraph, we define a method `chromosome-classes` for this class to tell GECO the number and classes of chromosomes which will be held by instances of this subclass of organism. Specifically, this method returns a list of length one (since we only need one chromosome), and the sole list element is the name of our application specific chromosome class, `binary-chromosome-10`.

```
(defclass SIMPLE-BINARY-10-ORGANISM (organism)
  ()
  (:documentation
   "An organism with only a 10-bit binary chromosome."))

(defmethod CHROMOSOME-CLASSES ((self simple-binary-10-organism))
  "So GECO will know what chromosomes to make."
  '(binary-chromosome-10))
```

The next class definition is a specialization of the GECO class `population-statistics`. Instances of this class are used by GECO to record statistical information about the current population to simplify certain operations like normalizing scores and determining whether the population has converged. By specializing this class, we'll be able to piggy-back some of the calculations we want performed on the functions which GECO will already be invoking. To record the additional information we want, we add a slot `allele-counts` and name the new specialized class `binary-population-statistics`. (This allele-count data isn't really useful for solving this particular problem. Adding it to the population's statistics was done here only to illustrate the use of an `:after` method to extend GECO's builtin functionality.)

```
(defclass BINARY-POPULATION-STATISTICS (population-statistics)
  ((ALLELE-COUNTS
    :accessor allele-counts
    :initform nil
    :type (or null (vector fixnum 10))
    :documentation
     "The number of non-zero alleles, by locus, for our population."))
  (:documentation
   "Our population-statistics also contains allele counts."))
```

The piggy-backed computation is performed by adding an `:after` method to the `compute-statistics` generic function, specialized on our `binary-population-statistics` class. This method sets the `allele-counts` slot to the result returned by invoking another GECO builtin function, `compute-binary-allele-statistics`, which returns a list of vectors (one per chromosome in the organisms of the population). Each vector contains counts of non-zero alleles, one count per locus. Since our organism only has one chromosome, we'll get a single vector of counts.

```
(defmethod COMPUTE-STATISTICS :AFTER
           ((pop-stats binary-population-statistics))
  "Compute the allele statistics for the population and save them."
  (setf (allele-counts pop-stats)
        (compute-binary-allele-statistics (population pop-stats))))
```

The next class definition is a specialization of the class `generational-population`, which is itself a specialization of the class `population`. The principle noteworthy feature of this subclass, `simple-binary-population`, is that it provides two additional methods. These methods provide information to GECO so that it can perform it's duties automatically. Specifically, the `organism-class` method tells GECO what class the organism instances are to be, and the `population-statistics-class` method tells GECO what class the population statistics instances are to be.

```
(defclass SIMPLE-BINARY-POPULATION
          (generational-population maximizing-score-mixin)
  ()
  (:documentation
   "Our populations are generational, and the scores are maximized."))

(defmethod ORGANISM-CLASS ((self simple-binary-population))
  "So GECO knows how to make the organisms in our population."
  'simple-binary-10-organism)

(defmethod POPULATION-STATISTICS-CLASS
           ((self simple-binary-population))
  "So GECO knows how to make our population statistics instances."
  'binary-population-statistics)
```

At this point please notice that telling GECO to create an instance of the class `simple-binary-population` is sufficient, and that GECO can then create a complete population of organisms of the proper class, and that each organism will contain chromosomes of the proper class, and each chromosome will have a loci vector of the proper size and type, initialized to random alleles. Thus, the structures (at this level) which will be manipulated by our GA are completely specified. Next, we need to specify the plan which controls the GA.

4.3 Defining a Genetic Plan

The next class definition is a specialization of `genetic-plan`. As mentioned earlier, the genetic plan provides a strategy which determines how an ecosystem regenerates, *i.e.*, how new organisms are created from older organisms. This is the heart of the genetic algorithm. Subclasses of the class `genetic-plan` are primarily used to specialize methods which perform the actual processing of the GA, but we define one additional slot, `statistics`, which will allow us to record statistics about each generation in a list as the population evolves under the plan. Alternatively, we could have used a file, or a vector which was large enough to hold the maximum number of generations, but simplicity will be the guiding principle for our example. Also, since we will have two different genetic plans, for the two examples illustrated in the file `sb-test.lisp`, we define an intermediate abstract class `simple-plan` to allow us to share some of the functionality between the two genetic plans.

```
(defclass SIMPLE-PLAN (genetic-plan)
  ((STATISTICS
    :accessor statistics
    :initarg :statistics
    :initform nil       ; so we can push instances
    :documentation
     "A stack of population-statistics for all past populations."))
  (:documentation
    "Abstract class to allow method sharing for initialization & regeneration."))
```

First is an `evaluate` method specialized on both the `simple-plan` and `simple-binary-10-organism` classes. This is the method which calculates the raw (unnormalized) score of each organism which our plan evolves. In our specific problem, score is proportional to the number of set bits in the chromosome, and there just happens to be a GECO utility which we can use: `count-allele-codes`. Inspecting the code for this method in `chromosome-methods.lisp` reveals that it can be used to return the number of loci in the chromosome which have allele codes of `1`.

```
(defmethod EVALUATE ((self simple-binary-10-organism)
                     (plan simple-plan)
                     &AUX (chromosome (first (genotype self))))
   "The score for our organisms is the number of non-zero alleles."
  #+:mcl (declare (ignore plan))
  (setf (score self)
        (count-allele-codes chromosome 0 (size chromosome) 1)))
```

A few additional points worth noting about `evaluate`:

- This is the only place in our example GA which needs to interpret the genetic content of our application-specific organism.
- Often it is necessary to `decode` the genetic content of an organism, converting the `genotype` into an instance of the `phenotype` represented by the genotype. GECO provides for this by including the following:
  - A `genotype` slot is defined in the `organism` class.
  - A `phenotype` slot is defined in the `organism-phenotype-mixin` class.
  - A `decode` generic function is called in a `:before` method of `evaluate` specialized on the `organism-phenotype-mixin` and `genetic-plan` classes.
  - Since GECO cannot predefine a method for `decode`, any GAs using phenotypes must be sure to implement one for the application specific subclass of `organism` and `organism-phenotype-mixin`.
- Generally there is little reason for the plan to be an argument to this particular method, but it is part of the protocol for the `evaluate` generic function, which is also used at the `ecosystem` and `population` levels of our class hierarchy, and at these higher levels it may well be appropriate for the genetic plan to discriminate between alternate methods.

The next method defined in our example is `regenerate`, which is specialized on both the `simple-plan` and the `simple-binary-population` classes. The purpose of this method is to create a new (or revised) population based on the current population, using whatever strategy is specific to the genetic plan. There is a default method provided by GECO in `genetic-plan-methods.lisp` for subclasses of `generational-population`, but it is provided as a template, not a realistic example, since it simply copies random organisms from generation to generation (plus some simple bookkeeping). Our specialized version of `regenerate` replaces the random copying with a call to a new generic function `operate-on-population` which takes the current and new (but empty) populations as input, and updates the new population. The example will define two versions of `operate-on-population`, discriminated by subclasses of our `simple-plan`. The `regenerate` method also records the current population's statistics in the list in the plan's `statistics` slot. As mentioned earlier, it could have written some of the statistical information to a file for later analysis, or possibly used them to support the genetic plan.

```
(defmethod REGENERATE ((plan simple-plan)
                       (old-pop simple-binary-population)
                       &AUX
                       (new-pop (make-population (ecosystem old-pop)
                                                 (class-of old-pop)
                                                 :size (size old-pop))))
  "Create a new generation from the previous one, and record statistics."
  (setf (ecosystem new-pop) (ecosystem old-pop))
  ;; selectively reproduce, crossover, and mutate
  (operate-on-population plan old-pop new-pop)
  ;; record old-pop's statistics
  (push (statistics old-pop)     ; impractical for real-world problems
        (statistics plan))
  new-pop)
```

Now we are almost ready to define our alternate versions of `operate-on-population` which contain the distinguishing features of our two example GAs. To keep them separate, we define two subclasses of our `simple-plan`: `simple-plan-1` and `simple-plan-2` (we'll only examine `simple-plan-1` here). We also give these classes separate specialized methods to supply the probabilities with which we should apply the mutate and crossover operators, since these values may need to be different for the two plans.

```
(defclass SIMPLE-PLAN-1 (simple-plan)
  ())

(defmethod PROB-MUTATE ((self SIMPLE-PLAN-1))
  "This is the probability of mutating an organism, not a single locus as is often used."
  0.03)

(defmethod PROB-CROSS ((self SIMPLE-PLAN-1))
  "The probability of crossover for an organism."
  0.7)
```

The method `operate-on-population` for `simple-plan-1` uses a technique referred to by Brindle [Bri81, Gol89] as "stochastic remainder selection without replacement" (`stochastic-remainder-preselect`, page 61) to select one organism at a time from the old (current) population, then based on a random draw applies either a uniform crossover operator [Sys89, SD91, Dav91] with another member of the old population (selected randomly), a simple bit mutation operator, or simple reproduction, to supply members of the new population. The principle difference found in the `operate-on-population` for `simple-plan-2` is that the second organism used in crossover is also selected based on fitness, in stead of randomly.

```
(defmethod OPERATE-ON-POPULATION
           ((plan simple-plan-1) old-population new-population &AUX
            (new-organisms (organisms new-population))
            (p-cross (prob-cross plan))
            (p-mutate (+ p-cross (prob-mutate plan)))
            (orphan (make-instance (organism-class old-population))))
  "Apply the genetic operators on selected organisms from the old population."
  (let ((selector (stochastic-remainder-preselect old-population)))
    (do ((org1 (funcall selector) (funcall selector))
         org2
         (random# (geco-random 1.0) (geco-random 1.0))
         (i 0 (1+ i)))
        ((null org1))
      (cond
       ((> p-cross random#)
        (if (< 1 (hamming-distance
                  (first (genotype org1))
                  (first (genotype (setf org2 (pick-random-organism
                                                old-population))))))
            (uniform-cross-organisms
             org1 org2
             (setf (aref new-organisms i)
                   (copy-organism
                    org1 :new-population new-population))
             orphan) ;; a throw-away, not in any population so it can be GC'd
          ;; hamming distances <2 will produce eidetic offspring anyway,
          ;; so bypass crossover & evaluation
          (setf (aref new-organisms i)
                (copy-organism-with-score
                 org1 :new-population new-population))))
       ((> p-mutate random#)
        (mutate-organism
         (setf (aref new-organisms i)
               (copy-organism
                org1 :new-population new-population))))
       (T ;; copying the score bypasses the need for a redundant evaluate
        (setf (aref new-organisms i)
              (copy-organism-with-score
               org1 :new-population new-population)))))))
```

The remaining code in `sb-test.lisp` simply provides a test harness to repeatedly invoke the GAs, and accumulate performance information over a specified number of runs.

Chapter 5

The GECO Files

This section provides a brief overview of the GECO files. The files are discussed in groups, based on related type or content.

The first group of files provide documentation.

- `README`
  An overview of the GECO distribution, including abstract, copyright and authorship information, installation instructions, version history.

- `COPYING.LIB-2.0`
  A copy of the GNU Library General Public License, version 2.0, which describes the terms under which GECO is distributed. This document is a product of the Free Software Foundation, Inc., of Cambridge, Mass.

- `geco.ps`
  A copy of the GECO documentation (this document), in PostScript form.

The next group of files are related to the GECO system definition.

- `GECO.system`
  The system definition file. It contains the `defpackage` and `defsystem` forms for creating GECO, and code to select conditional compilation features. This is the only GECO file which is normally loaded manually. To compile and load the rest of GECO, use the example commands contained in comments following the `defsystem` forms.

- `defsystem.lisp`
  A portable defsystem facility, developed by Mark Kantrowitz, School of Computer Science, Carnegie Mellon University. This is the defsystem used by `GECO.system`. This

file is a slightly modified version based on one obtained from the directory `/afs/cs.cmu.edu/project/ai-repository/ai/lang/lisp` via anonymous FTP from `ftp.cs.cmu.edu`.[1] The modifications allow it to work under MCL 2.0, and under Franz's Allegro Common Lisp versions prior to the patched 4.1 which supports logical pathnames.

- `defsystem.text`
  Provides the documentation for `defsystem.lisp`. This version was obtained from the directory `/afs/cs.cmu.edu/project/ai-repository/ai/lang/lisp` via anonymous FTP from `ftp.cs.cmu.edu`.

The next group of files contain definitions which must be loaded/compiled before the rest of the GECO source code.

- `generics.lisp`
  This file contains `defgeneric` forms defining some (but not all) of the generic function *protocol* established by GECO, *i.e.*, the set of generic functions and their arguments which must be honored by all GECO-based applications. Each of the `defgeneric` forms contains a `:documentation` string for the function describing its intended purpose (these documentation strings are easily retrieved in most interactive lisp programming environments). Comments in the file also indicate which of the generic functions should/must have methods defined for your application-specific classes when you implement a GA with GECO.

- `classes.lisp`
  This file contains the `defclass` forms defining each of the GECO classes.

- `dbg.lisp`
  This file contains the definitions for a general debugging facility used in the development of GECO.

- `random.lisp`
  This file contains the definition of the random number generators used by GECO, `geco-random-integer` and `geco-random-float`. In addition, it includes the definition of an alternate set of random number generators, provided with permission from John Koza from his implementation Simple Genetic Programming in Lisp. Conditional compilation options (setup in `GECO.system`) control which random number generator is used.

[1]This location is actually different from the one from which I originally obtained this software, but this is the latest address (of which I am aware at the time of this writing) of Mark Kantrowitz's archive. This archive seems to be reorganized frequently, but last time I checked, the `defsystem` files were in the `code/tools` subdirectory.

The next group of files will eventually be used to support a more sophisticated memory management scheme than is presently being used by GECO. The corresponding defsystem entries are commented out, but these files are provided in case an ambitious GECO user should want to pursue this enhancement (please let me know if you do!).

- `bwm-resources.lisp`
  A portable resources facility, developed by Bradford W. Miller, Department of Computer Science, University of Rochester. This file contains its own documentation. This version was obtained via anonymous FTP as `resources.lisp` from `ftp.cs.cmu.edu` in the directory `/afs/cs.cmu.edu/project/ai-repository/ai/lang/lisp`.[2]

- `resource-mgt.lisp`
  This file contains some GECO specific resource management tools built on top of `bwm-resources.lisp`. These tools are presently relatively untested.

The remaining files contain the method definitions for the guts of GECO. An attempt has been made to organize them by the principle class to which the methods apply, however, due to the use of multiple-dispatch methods, this has not always been possible.

In general, the files have been named using a standard pattern: *class-name*`-methods.lisp`. Presently, the single exception is the file `selection-methods.lisp` which I decided to separate from the other `population` methods.

- `ecosystem-methods.lisp`
  This file contains methods which perform the following general categories of operations:
  - initialize ecosystems
  - make instances of `population` and `genetic-plan` appropriate for an ecosystem
  - evolve and evaluate ecosystems

- `genetic-plan-methods.lisp`
  This file contains methods which perform the following general categories of operations:
  - regenerate instances of `ecosystem` and `population`
  - determine whether evolution should be terminated

[2]As with the `defsystem` files, this is a different location that from which I originally obtained this software. My most recent information regarding its location within this archive pointed to the `code/ext/resource` subdirectory.

- `population-methods.lisp`
  This file contains methods which perform the following general categories of operations:
  - initialize and print instances of `population`
  - create `organism` and `population-statistics` instances for a population
  - evaluate populations, and compute statistics over them
  - compute normalized scores over populations
  - determine if a population has converged
- `pop-stats-methods.lisp`
  This file contains methods which perform the following general categories of operations:
  - initialize, and print instances of `population-statistics`
  - compute and normalize population statistics
- `selection-methods.lisp`
  A fairly broad sampling of techniques for selecting organisms from populations. Techniques include:
  - random selection
  - weighted roulette-wheel selection
  - stochastic remainder selection
  - tournament selection

  A version of the roulette-wheel selection routine has also been generalized to select an index from a table of weights. I expect this routine to be useful for performing weighted genetic operator selection.
- `organism-methods.lisp`
  This file contains methods which perform the following general categories of operations:
  - initialize, copy, and print instances of `organism`
  - create chromosomes for an organism
  - evaluate and decode organisms
  - compute normalized scores of organisms
  - determine if two organisms are the same
  - choose random chromosomes, and locations on chromosomes

  - perform mutation and crossover on organisms

- `chromosome-methods.lisp`
  This file contains methods which perform the following general categories of operations:
  - initialize, copy, and print instances of `chromosome`
  - create and print loci vectors
  - access individual loci
  - pick random loci and allele values
  - count allele values
  - convert (internal) allele codes to (printable) allele values
  - decode binary chromosomes (including gray coded representations)
  - determine if two chromosomes are the same
  - determine the Hamming distance between two chromosomes
  - perform mutation and crossover on chromosomes

There are also two files containing example GAs. These files aren't intended to show impressive solutions to tough problems; rather they are intended to show how one might go about building a GA using GECO.

- `sb-test.lisp`
  This is the simple binary example discussed in Chapter 4.
- `ss-test.lisp`
  This is another simple example, using a sequence-based chromosome.

Bibliography

[Ack87] D. A. Ackley. *A Connectionist Machine for Genetic Hillclimbing.* Kluwer Academic Publishers, 1987.

[Bak85] J. E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 100–101. Lawrence Erlbaum Associates, 1985.

[BHS91] Thomas Bäck, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann, 1991.

[Bri81] A. Brindle. Genetic algorithms for function optimization, 1981. Unpublished doctoral dissertation, University of Alberta, Edmonton.

[CS88] R. A. Caruna and J. D. Schaffer. Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In J. Laird, editor, *Proceedings of the Fifth International Conference on Machine Learning*, pages 153–161. Morgan Kaufmann, June 1988.

[Dav91] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, 1991.

[DeJ75] K. A. DeJong. *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan, 1975.

[Gol82] David E. Goldberg. SGA: A simple genetic algorithm. Computer program in Pascal, University of Michigan, Department of Civil Engineering, Ann Arbor, 1982.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Academic Press, The University of Alabama, 1989.

[Gre84a] J. J. Grefenstette. GENESIS: A system for using genetic search procedures. In *Proceedings of the 1984 Conference on Intelligent Systems and Machines*, pages 161–165, 1984.

[Gre84b] J. J. Grefenstette. A user's guide to GENESIS. Technical Report CS-84-11, Vanderbilt University, Department of Computer Science, Nashville, 1984.

[HHNT87] John H. Holland, K. J. Holyoak, R. E. Nisbett, and P. R. Thagard. *Induction: Processes of Inference, Learning, and Discovery.* The MIT Press, Cambridge, 1987.

[Hol75] John H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems.* The MIT Press, 1992. This is a revised and extended version of [Hol75].

[HR78] John H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In D. A. Waterman and F. Hayes-Roth, editors, *Pattern Directed Inference Systems*, pages 313–329. Addison-Wesley, New York, 1978.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Academic Press, 1989.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection and Genetics.* The MIT Press, June 1992.

[MJ91] Z. Michalewicz and C. Janikow. Data structures + genetic operators = evolution programs. Technical report, UNCC, 1991.

[Rad92a] Nicholas J. Radcliffe. The algebra of genetic algorithms. Technical Report EPCC-92-11, Edinburgh Parallel Computing Centre, University of Edinburgh, 1992.

[Rad92b] Nicholas J. Radcliffe. Non-linear genetic representations. In *Parallel Problem Solving from Nature 2*. Elsevier Science Publishers, 1992.

[SD91] William M. Spears and Kenneth A. DeJong. On the virtues of parameterised uniform crossover. In Richard K. Belew and Lashon B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236. Morgan Kaufmann, 1991.

[Spe91] William M. Spears. GAL. Computer program in Lisp, Navy Center for Applied Research in AI, Naval Research Laboratory, 1991.

[Ste90] Guy L. Steele, Jr. *Common Lisp, The Language*. Digital Press, second edition, 1990.

[Sys89] Gilbert Syswerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[WK88] D. Whitley and J. Kauth. GENITOR: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, 1988.

Index