
Daan van Beek
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
d.i.vanbeek@student.utwente.nl

ABSTRACT
By studying single-objective genetic algorithms and multi-objective genetic algorithms this paper determines the functions needed to update existing single-objective genetic algorithms programmed in functional languages in order to make them applicable to multi-objective problems. By performing a literature study knowledge about genetic algorithms and their special multi-objective versions was collected and existing single-objective genetic algorithm frameworks where examined for missing functionality. Using this knowledge a way was derived to update single-objective genetic algorithm frameworks. With the results of this paper one is able to update a single-objective framework in order to make it applicable to multi-objective problems.

1. INTRODUCTION
The group of genetic algorithms consists of algorithms that use Darwinian evolution as a search technique. Using this search technique one can substitute possibly non existing dedicated solutions, making genetic algorithms useful for searching through the solution space of NP-Complete problems. The group of Multi-objective genetic algorithms is a group of algorithms that are specially fitted to handle multiple connected problems at once. These multiple problems can for example share resources.

As genetic algorithms use only some problem specific knowledge, implementing one has been made easy by creating genetic algorithm frameworks. A genetic algorithm can be created by giving some problem specific information to such a framework and executing the framework. These frameworks consists of the logic needed to use Darwinian evolution to find solutions to the problem specified by the user.

In the field of functional programming languages, a number of genetic algorithm frameworks have already been created. Note that none of these frameworks are applicable to multi-objective problems. This paper researches the fields of genetic algorithms and multi-objective genetic algorithms, and looks into existing functional frameworks in order to determine what functions are needed to update existing functional frameworks to make them applicable to multi-objective problems. The needed research is done by performing a literature study, looking into (multi objective) genetic algorithms in general and in a functional programming context. The functions needed to update existing frameworks follow from the literature study.

In order to determine the needed functions a basic understanding of genetic algorithms and multi-objective algorithms is needed. Chapter 2 discusses genetic algorithms, what problems they solve and how they are implemented. This discussion of genetic algorithms is concluded with an example problem (Traveling Salesman Problem). Chapter 3 discusses Multi-objective genetic algorithms. As for genetic algorithms it is first discussed what problems they solve and then how they are implemented. Chapter 4 gives a short discussion of the use of functional programming languages in the field of genetic algorithms and in chapter 5 conclusions are drawn and the needed functions are specified, according to the theory in Chapters 2 and 3. Also an example is given of how an existing functional framework can be made applicable to multi-objective problems.

2. GENETIC ALGORITHMS
In order to understand genetic algorithms paragraph 2.1 describes what problems they solve. Afterward paragraph 2.2 looks at the standard implementation of genetic algorithms. Paragraph 2.3 contains an example to see how theory is applied in practice.

2.1 Problems solved by genetic algorithms
Genetic algorithms can solve all kinds of optimization problems. A lot of problems have known algorithms that are fast and efficient. Although a genetic algorithm can be used here it will not outperform the problem a specific algorithm as it uses almost no problem specific knowledge. Genetic algorithms are best known for their solutions to intractable problems/NP-Complete problems for which no efficient specific algorithm exist [7, 10]. The applicability of genetic algorithms is vast because of the minimal problem specific knowledge used to solve the problem [2, 8, 13]. The group of genetic algorithms consist of algorithms that use the principles of Darwinian "survival of the fittest" evolution [1, 4]. A search using Darwinian evolution as its search technique is possibly best described as a "strongly directed random search".

2.2 General implementation of genetic algorithms
The search technique employed by genetic algorithms is a very general one that does not use a lot of problem specific information. Because of this generality, it is possible to describe genetic algorithms with a standard structure that has only to be adapted at certain known points to provide an algorithm for a specific problem. The structure itself is explained in paragraph 2.2.1. The concept of chromosomes, the fitness function and the functions for selection breeding and merging are discussed in paragraphs 2.2.2 to 2.2.6. Paragraphs 2.2.7 to 2.2.10 discusses some problems and their solutions that have to do with the inner workings of genetic algorithms.
2.2.1 General procedure of genetic algorithms
Provided the genetic representation of a candidate solution in chromosomes and the fitness function (both explained later), a genetic algorithm is able to find a solution to the problem in the following way:

Once:
- Create an initial population by randomly generating a set of candidate solutions (chromosomes).

Repeat:
- Selection: Use the fitness function to determine which chromosomes from the current population will be used for breeding (selecting the parents).
- Breeding: Create new child candidate solutions by using genetic operators on the selected parents.
- Merging: Combine the new child solutions with the old population of candidate solutions to create the new current population.

2.2.2 Chromosomes
In order to tailor a genetic algorithm to a specific problem a representation of a candidate solution has to be provided. This representation is called a chromosome. Such a chromosome describes everything that defines a candidate solution such as all the parameters that describe an answer to the problem. Most of the time chromosomes take on the form of a simple list of items. These items are called genes [7]. The reason these kind of representations are needed is because all of the genetic functions of genetic algorithm frameworks are only capable to handle chromosomes and genes. This way none of the non-problem specific functions have to be rewritten when the genetic algorithm has to solve a different problem.

2.2.3 Fitness function
A genetic algorithm also needs a way to determine if one candidate solution is better than another. This information is given to the genetic algorithm by specifying a fitness function that rates candidate solutions from worst to best. The ratings of candidate solutions are used by the genetic algorithm to direct the search for the best solution [1, 4, 7, 11]. A mathematical function is not always able to rate the candidate solutions such as pictures. In those cases the candidate solutions could be displayed on screen and an user could be asked to provide a rating [2].

2.2.4 Selection
The reason why a genetic algorithm work is because they move the population of candidate solutions towards an optimum. A high probability of getting better solutions from worse solutions (child from parents) is achieved by careful selection of the chromosomes that will be used in breeding.

It may come as a surprise that not only the best chromosomes are selected. Some worse or bad solutions may also be used for breeding in order to keep the diversity of the population intact. Keeping a diverse population can help the overall search overcome local optima by keeping alternate search directions alive. If all bad or worse solutions are deleted the search becomes pointed to one search direction which might not lead to the best answer [12].

2.2.5 Breeding
In order to create new child chromosomes from parent chromosomes multiple genetic operators can be used. These functions can be divided into mutation functions and crossover functions. Not all breeding functions are applicable to all problems, sometimes a new breeding function for some type of chromosomes will have to be specified. Most genetic algorithm frameworks provide a library of genetic operators applicable to different types of chromosome representations such as simple lists of items and bit strings [1, 4].

Mutation functions are functions that use one parent chromosome to create one child chromosome. In order to do this one or multiple genes of a parent can be randomly changed or some genes can be swapped [13]. Swapping genes is especially applicable to problems that are order-based such as the traveling salesman problem. An example of an swapping genes function is the Allele Swap [7].

Crossover function use two parents to create new child chromosomes(s). An example is one 1-Point crossover where the chromosomes of two parents are split in the same position and are put together with the other half of the other parent to create two new child chromosomes [1, 13].

The occurrence of mutation and crossover operations are subject to chance. Making the occurrence of these operators subject to chance mimics the stochastic nature of evolution [13]. The chance of occurrence can be set by parameters that are subject to testing.

2.2.6 Merging
The merging part of the algorithm is responsible for merging the current population with the newly created child chromosomes. A lot of choices can be made on how to do this and these choices can have a great influence on the resulting efficiency of the algorithm. Two common types of merging functions are "Stable State" and "Generational" [1, 7].

When "Stable State" is used as the merging function of the genetic algorithm, merging is performed by replacing the worst chromosomes from the current population with the new child chromosomes. Variables such as how much child chromosomes are to be generated in this case are subject to testing. "Generational" means that the current chromosomes are only used to create child chromosomes. Only the child chromosomes are used to construct the new population [7].

When choosing the merging function it is important to keep a close eye to the diversity of the resulting population. It could be wise to use some randomness in the merging function as to give chromosomes with a low fitness value a chance to appear in the new population [7].

2.2.7 Termination
For some problems it is possible to find the best solution. This solution will have the maximum fitness value. If found, the algorithm can be stopped and the best solution may be returned. Other termination conditions may be time or computational constraints [4]. In that case the algorithm will have to return the solution in the current population that has the highest fitness value. It is also possible for a genetic algorithm to stop when no progress has been made for a predefined number of rounds or if a satisfying solution has been found [13].

2.2.8 Fast Convergence
As genetic algorithms tend to favor the best solution found so far, it is possible that the candidate solution set becomes very monotone early in the search. This phenomenon is called Fast Convergence. By having a non-diverse solution set, the possibility of finding only a local optimum increases. If the solution set is kept diverse it is possible to reach beyond local optima. In order to prevent fast convergence it is possible to inject a few randomly generated candidate solutions into the solution set at the beginning of selection or merging. This way a few special possibly totally different solutions always
influence the search and possibly guide the search towards a better solution.

2.2.9 Tweaking a genetic algorithm
The genetic representation of candidate solutions (chromosomes) and the fitness function are not the only things the user has to specify before a genetic algorithm can be used. The user also needs to choose what genetic operators to use for this specific problem. Genetic operators are the functions for selection, breeding and merging. There are also some environmental parameters of the genetic algorithm such as the population size and some settings of the chosen functions for selection, breeding and merging. Environmental parameters can be for example "chances of things happening" such as the cross-over rate or the mutation rate [4].

2.2.10 Elitism
Genetic Algorithms search through the solution space by applying random changes to already found solutions. Because of this search method there exists a possibility that already found good solutions get lost in the process [3, 13]. Elitism is a mechanism proposed to counter this effect. When using elitism the algorithm keeps an external population of candidate solutions that are not dominated by any other solutions. This external population can be used in the selection process and or in the process of combining populations to create the new population. It is also possible that the external population does not participate in the selection or combining procedures at all. The external population is often called "archive" [13]. What is important is that a solution in the elitist group is only thrown away when it is replaced by newly found solution that dominates this solution. In this way elitism makes sure the best solutions found are preserved [5].

2.3 Example - Traveling Salesman Problem
The Traveling Salesman Problem (TSP) is used as an example in many articles [1, 7]. The TSP problem is very suitable for genetic algorithms not only because of its NP-Completeness but also because most frameworks have genetic operators for problems that have order-based chromosomes. In this paragraph TSP will be used to give an example of how a genetic algorithm works on a specific problem. Other examples can be found in [1, 7].

2.3.1 The TSP Problem
The Traveling Salesman Problem is about finding an minimal Hamilton cycle between a number of cities. The given problem consists of a number of cities and information about the cost of travel between each pair of cities. This cost of travel could be based on real world distance, allowed speed or any other condition that defines one connection between cities to be better than another. In plain English the TSP is to find a path that connects all given cities, visiting none of the cities multiple times. The path must reconnect to the starting point and the path must be minimal in regard to the given cost of each used connection between a pair of cities.

2.3.2 Chromosome representation and fitness function
A chromosome is a collection of information, that uniquely defines a candidate solution. All candidate solutions must be expressible in this collection of information. The single information bits are called genes. An answer to a TSP problem consists of the generated optimal path. This path can be described by specifying an ordering of all the cities. This suggests a chromosome should take the form of a list, containing the specific order in which the cities are visited. For simplicity all the cities can be assigned an integer in the range of 1 and the number of cities. Chromosomes will be a list of integers that express the order proposed by this candidate solution. As every city can only be visited once, it should be noted that every integer can occur only once in a chromosome and that all integers (cities) have to be used.

The fitness function for the TSP is simply the accumulated cost of all paths taken as described by the chromosome currently evaluated. In this case, a lower value from the fitness function means a better solution. The cost of traveling back from the last city to the first should also be included.

2.3.3 Choice of selection, breeding and merging functions
The selection function is rarely problem specific. As this is also the case for the TSP problem, normal framework functions can be used. The breeding function does need special attention as some functions do not produce valid chromosomes. A chromosome cannot contain one city more than once. It is also true that the integer value of a gene is not of importance (as in less or more). Only the gene ordering is of importance.

Due to the above considerations the simple mutation functions that change values of single genes cannot be used for breeding. Using crossover on two parent chromosomes would also rarely result in valid child chromosomes. The Allele Swap mutation operation is applicable, though. This mutation operation will swap some genes from a parent chromosome to create a child chromosome. In the case of TSP this means that the route is changed by switching the position of one city with the position of another city [7].

The merging functions can also be used from standard frameworks as they, just as the selection functions, do not have any specific problems with the type of chromosomes used.

2.3.4 TSP - Why genetic algorithms work
After specifying the chromosomal representation, the fitness function, the selection, breeding and merging functions the algorithm is ready to run.

It might seem strange that a genetic algorithm is able to generate a near optimal solution while it uses a lot of randomness. This may be partially attributed to the selection process. By selecting the right parents the search becomes directed towards a (local) optimum. By also keeping or introducing some bad candidate solutions genetic algorithms are able to reach further than local optima towards the real optimum.

3. MULTI-OBJECTIVE GENETIC ALGORITHMS
Now that a general understanding of genetic algorithms has been established, we can investigate multi-objective genetic algorithms. Analogues to chapter 2, paragraph 3.1 describes the problems solved by multi-objective genetic algorithms. Paragraph 3.2 describes the general implementation and Paragraph 3.3 describes an example implementation. We will see that implementation of a multi-objective genetic algorithms differs in just a couple areas in respect to "normal" genetic algorithms. These differences are examined with extra care as are the exact functions we need to augment the existing frameworks with in order to make them applicable to multi-objective optimization problems.
3.1 Problems solved by multi-objective genetic algorithms

Multi-objective optimization problems are problems that have multiple criterion/objectives to be optimized. These objectives share resources and cannot all be optimized to their full individual potential because of underlying constraints. Genetic algorithms can be adapted to solve this kind of problems. MOO genetic algorithms are very similar to normal genetic algorithms as they are especially applicable to NP-Complete problems. Because of the tradeoff between the multiple objectives it almost always the case that multiple optimized solutions exist [3]. The objectives are as follows [10, 12-13]:

- Search for a solution that optimizes all objectives together.
- Finding multiple solutions is preferred.
- The multiple optimal solutions should be as diverse as possible.

A set of solutions to a multiple objective optimization problem that are all different but all optimal is called a Pareto optimal solution set [12]. All solutions in a Pareto optimal solution set are non-dominated solutions. The term non-dominated comes from the principle of Pareto dominance; The principle of Pareto dominance is as follows: Solution A dominates solution B when A is a better solution than B in at least one sub objective, and not a worse solution in any sub objective. In other words, solution A is a better solution at at least one objective or just an as good solution at all other objectives as solution B [10, 13].

Because of the tradeoffs existing between the multiple sub problems it is possible that multiple answers are optimal but still different. Generating multiple answers is preferable because the user can then decide between all optimal solutions which tradeoff or compromise to accept. Ensuring a diverse Pareto optimal set makes sure that there is something to choose from [3, 13]. If all answers found would just in essence be the same (slight differences but same sub objectives compromised) it would be just as valuable to generate just one such an answer.

Multi-objective optimization problems can be solved by genetic algorithms using multiple strategies which are discussed next.

3.2 General implementation of multi-objective genetic algorithms

Multiple-objective optimization genetic algorithms (MOOGA) can be solved using different techniques and strategies. Fitness assignment techniques have to do with ranking candidate solutions for MOO's. The section about diversity discusses techniques that make sure that the solution set found is as diverse as possible.

3.2.1 Fitness assignment techniques

The reason why MOO problems cannot be solved by normal GA's is that an overall fitness function usually is not available [3, 10]. All sub-problems have their own fitness functions and these can be used in different ways to rate the overall problem. For all techniques described here there exist many variations in the detailed implementations [10].

3.2.1.1 Aggregation

Aggregation techniques attempt to combine the fitness functions of the separate sub-problems in order to construct one overall fitness function. This fitness function can be for example computed by taking a weighted sum of all the fitness functions of the sub problems [10, 13]. This overall fitness function can subsequently be used in the original single objective genetic algorithms.

A drawback from this method is that the weighted sums have to be specified which can be difficult when the underlying relations between the sub problems is not clear [10]. Another issue is that the overall fitness function implies a preferred tradeoff between the sub problems and so the genetic algorithm will not generate a diverse Pareto optimal solution set. One way to eliminate this problem is to systematically varying the weights of the weighted sum function [13].

3.2.1.2 Criterion

A criterion based approach uses one of the available fitness functions from the sub problems to make decisions for selection. By consequently rotating the used fitness function they all exert influence in the selected population for mating [13]. The drawback from this solution is that GA's that use this technique tend to generate solutions in which one sub problem is very optimal while the rest of the sub problems are heavily compromised [10].

3.2.1.3 Pareto dominance

An approach that is better able to optimize multiple objectives at once are techniques using the principle of Pareto dominance. Although all MOOGA's search for the Pareto optimal set, it is also possible to use this principle as a fitness function. In order to determine the fitness of solution A it is possible to count the number of solutions dominated by A (dominance rank) or the number of solution that A dominates (dominance count) [10, 13].

More involved is the method of dominance fronts/depth where multiple Pareto optimal fronts are detected by repeating these steps: find all non-dominated solutions, call the non-dominated solutions front 1, remove them from the population and repeat to create front 2 etc. The depth information can then be used as a fitness function [10, 13].

3.2.2 Diversity

In order to generate a diverse Pareto optimal set it is important to be able to collect density information about the current solution set. The density of a certain candidate solution is information about how much almost similar candidate solutions are in the solution set. This information can be used in the selection process to stimulate searching in areas that have a low density in order to keep the solution set diverse. It is also possible to throw away solutions in high density areas when applying the merging function.

Using density information in the selection procedure is often implemented by the concept of fitness sharing. This concept simply states that it is possible to include density information in the fitness function so that the fitness value of solutions in dense areas are negatively influenced [10, 13].

One way to determine density information is by applying kernel methods. These functions calculate the distances from this solution to all other solutions and sums up all these distances. Solutions with a higher value are preferred. Other strategies are also possible such as using the distance to the nearest other solution as an indication of the density. The total number of solutions in a certain solution are can also be used as a density indication [13].

As most genetic algorithms use elitism, it is also possible to use the external set of non-dominated solutions, called the archive, as an indicator of density for a certain solution area. When for example the merging function is applied, a
preference can be given to solutions that are in the least crowded areas of the archive [13].

3.3 Example - Traveling Salesman Problem with multiple objectives

In the chapter about normal genetic algorithms the TSP problem was used to clarify some concepts. The TSP problem is also usable as an example of a multiple objective optimization problem in the following way:

3.3.1 Expanding the TSP problem

Let's assume that when we last calculated the optimal Hamilton cycle between the city's we used the distance between two city's as the cost made when traveling from one city to another. Looking more closely to the problem we discover that city's also have an elevation. When two cities are not on the same elevation level it will cost a traveling salesman the same or more gas to travel between these two city's because he has to overcome the height difference. It also just so happens that salesman's car has no feature to save or store energy when driving from high elevation to low elevation.

As the salesman has to pay for the gasoline usage, he wants a path that has the least possible accumulated elevation differences from low elevation to high elevation city's. He also still wants a Hamilton cycle that has the least accumulated distance between the visited city's in order to be fast.

As there are now two objectives that need to be optimized: distance and gasoline usage, a Genetic multi objective optimization algorithm can be used. This way the salesman does not need to know anything about the preferred tradeoff between distance and extra gasoline usage due to height. When using an MOOGA the salesman will also be able to choose between different optimal solutions that represent different tradeoffs [3, 13]. He could for example choose a solution that uses less gasoline but has more traveling time. If a short traveling time is more important than gasoline usage he could choose an optimal solution that favors fast travel time.

3.3.2 Chromosome representation and fitness function

Although the problem has changed significantly, the chromosome representation has not. A candidate solution to the TSP problem can still be described as a list of cities, representing the order in which they are visited.

The fitness function will now have to incorporate both objectives in its rating of candidate solutions. As described in this chapter a good way to do this is using the concept of Pareto Dominance. One of multiple implementations can be used such as the dominance rank or dominance count.

3.3.3 Choice of selection, breeding and merging functions

In order to ensure the diversity of the solution set the selection and/or merging functions used will have to incorporate density information. This way the selection and/or merging functions will favor the solutions that represent an optimum that is not properly represented in the solution set. Using density information will increase the chance that the salesman can choose between different but optimal solutions.

3.3.4 Algorithm tweaks

As with all genetic algorithms a lot of tweaking can be done. If it is apparent that the algorithm does not produce diverse answers it could be useful to add some random solutions at the beginning of each round in order to inspire the algorithm to explore alternatives. Choices such as: where in the algorithm to use density information, where to use the archive and what fitness assignment technique, can be changed according to the achieved results. As multiple papers have proven the concept of Elitism effective, it should always be used [3]. Its implementation could also help with providing more density information.

4. COMPARING A FUNCTIONAL WITH IMPERATIVE APPROACHES

This next section of the article is specifically about genetic algorithms programmed in functional languages. The choice for functional languages is not an arbitrary one. This section of the article explains what are the pros and cons for implementing genetic algorithms in a functional language in comparison to a normal imperative language.

4.1 Ease of implementation

Multiple research communities have chosen Haskell as the implementation language for their genetic algorithms [1, 4, 7, 11]. They mostly justify this by noting that the functional nature of a genetic algorithm makes it perfect for a functional language. As can be read in the chapters above, a functional algorithm uses a couple of functions such as a selection, mutation and a breeding function. By describing these functions in Haskell and using function composition to make the library functions for selection, mutation and breeding interchangeable, easy to use frameworks can be implemented. When using a functional language, parameters given to the algorithm can also consist of functions, such as extra or overriding functions for special problems. This also makes a functional implementation very flexible [7]. Although these functionalities can also be created in most imperative languages, they occur naturally in most functional languages which makes the implementation process more natural and easier to understand. Lazy evaluation can also be used to the advantage of the genetic algorithm by expressing the breeding function as an infinite list. Because of the lazy evaluation, only new child solutions that are currently needed would really be evaluated. Although infinitely more are described by the merging function [1].

4.2 Execution speeds

Due to the functional nature of a functional language, creating a genetic algorithm (framework) will take less time in a functional language than implementing it in an imperative language. For now, the functional version of the framework described in [4] is outperformed by the imperative version, but the paper also states that with proper tools these performance problems should be solvable.

5. FUNCTIONAL GENETIC ALGORITHM FRAMEWORKS

In order to specify the functions needed to augment the existing frameworks in order to make them applicable to multi-objective optimization problems, we will first state what functional frameworks already exist and what functionality they offer. After investigating the existing functional frameworks we will specify the needed functions using the existing frameworks and the knowledge collected in chapter 2 and 3 of this paper.

5.1 Currently existing frameworks

As noted earlier no functional frameworks where found with any special multi-objective optimization abilities. Multiple functional frameworks where found that support fast
implementation of single-objective genetic algorithms, with all minor differences.

A genetic algorithm framework using Haskell [1]
The paper first states the general procedure follow by genetic algorithms and then gives an example of the used Breed function. Although more functions are supported, these are the ones named in their article: Crossover: 1-Point, order. Mutate: bit wise, one position. Selection: roulette-wheel, tournament, linear ranking. Combine (Merge): Generational, steady state. An example of the "onePointCross" function is provided which is a crossover function. Further in the article it is mentioned they also implemented the function "generelitism", which is a repopulation strategy that uses elitism.

A functional framework for the implementation of genetic...[4]
This framework consists of several modules, such as a general genetic algorithm that handles selection, breeding etc and a library of genetic operators that can be used to tweak the algorithm to the needs of the specific problem. These operators are functions used for selection, crossover and mutation. The article also give an overview of the precise user defined modules needed to run a genetic algorithm.

A Generic Functional Genetic Algorithm [7]
Except providing functionality for the general procedure of a genetic functions, this framework also supports the genetic operator "allele swap" (useful for TSP). One-point Crossover, Tournament Selection and Stable State merging are also given extra attention in the article.

Genetic Algorithms In Haskell with polytypic programming [11]
This article is very elaborate when it comes to the implementation and is about genetic programming, a method to create plans or programs using Darwinian evolution as a search method. A lot of code examples can be found in this paper. Some code examples use polytypic programming as stated in the title.

5.2 Needed functions to expand existing frameworks
Now that we have seen what functions are already provided by the existing frameworks and what functions are needed to support multi-objective genetic algorithms, we can conclude the following:
- Implementations of all standard genetic operators for selection, breeding and merging already exist.
- Elitism is also available in one framework.
- Multiple fitness assignment techniques for multi-objective problems have to be implemented.
- In order to find multiple different optimal solutions, density information has to be incorporated in the overall fitness function and/or in an updated merging function. This mechanism may also use the elitism archive (when available) to check what candidate solutions are preferred.
- Making an existing framework usable for multi-objective problems means updating the fitness function and maybe the merging function.

Updating existing functional frameworks can be done by updating the selection procedure to get its fitness values from a new overall fitness function that combines all user given fitness functions and density information. Using this new overall fitness function the frameworks would be able to find diverse Pareto optimal solution sets.

The method to convert single objective algorithm frameworks into frameworks that are capable of multi-objective algorithms consists of the following steps:
- Adjust the configuration options/files in order to give the user the possibility to specify multiple fitness functions.
- Add a new overall fitness function to the framework that has as its input: all the fitness functions, the candidate solution to be evaluated and all known candidate solutions. And as its output: a fitness value of the evaluated candidate solution based on its Pareto dominance over the rest of the solutions and its density information in respect to the other solutions.
- Adjust the selection function(s) in order to use the new overall fitness function.

5.3 Case Study: Updating an existing functional framework
HackageDB [6], which is a source for Haskell libraries, contains a genetic programming framework only usable for non multi-objective genetic algorithms [5]. It also contains a possibly usable but more specific framework for genetic programming [9]. In order to validate the conclusions reached in the above paragraph, we will now show how the non multi-objective genetic algorithm "hgalib" [5] can be transformed to handle multi-objective problems.

5.3.1 Providing fitness functions by user
In the example functional framework "hgalib" the user is asked to provide the single fitness function by specifying the function in a data construct before as part of the framework setup. This data construct has the following form:

```
data ChromosomeConfig c p =
  ChromosomeConfig {
    fitness :: c -> Double,
    * other function specifications *
  }
```

As the framework now has to support multiple objectives and multiple fitness functions, this user specification of the problem has to be adapted to support specifying multiple fitness functions. This can be done by using a list of fitness functions:

```
data ChromosomeConfig c p =
  ChromosomeConfig {
    fitnessFunctions :: [c -> Double],
    * other function specifications *
  }
```

5.3.2 Creating a new overall fitness function
Specifying a overall fitness function involves choosing what fitness assignment technique to use. As explained multiple options exist, categorized in Aggregation, Criterion and Pareto Dominance. Because of the modular nature of the examined frameworks, it is possible to create multiple overall fitness functions and make the user choose which one to use when setting up the framework configuration. The configuration data construct would then look as follows:
data ChromosomeConfig c p =
    ChromosomeConfig {
        fitnessFunctions :: [c -> Double],
        fitness :: c -> Double,
        * other function specifications *
    }

In order to give users a choice to what fitness assignment technique to use multiple overall fitness functions should be implemented in a general library. This task has been included in the future work section. Examples of usable multiple objective fitness assignment algorithms can be found in [13].

5.3.3 Incorporating the new overall fitness function into the framework
Incorporating the function can be accomplished by updating all selection functions to use the new overall fitness function. In this particular example this is not even needed, as we gave the overall fitness function the same name in the algorithm configuration as the single fitness function had.

6. CONCLUSIONS
This paper has shown how to augment existing functional single-objective genetic algorithm frameworks in order to make them applicable to multi-objective problems. By first researching the fields of genetic algorithms and multi-objective genetic algorithms and by determining what functionality the existing functional frameworks already have, we were able to determine the functions described in chapter 5 that enable single-objective frameworks to solve multi-objective problems. Future work could consist of actually updating some framework, preferably "hgalib"[5] in order to provide a functional multi-objective solution to the HackageDB [6] library.

7. REFERENCES