Genejector

Genetic programming with reflection and injection

March 2013

University of Southern Denmark
Department of Mathematics and Computer Science

Bachelor thesis by
Christian Funder Sommerlund
zero3@zero3.dk

Supervisor
Peter Schneider-Kamp
petersk@imada.sdu.dk

Abstract

The field of genetic programming has been around longer than most programming languages, but still seems largely based on decade-old methods. I argue that modern programming concepts such as *reflection*, *data structures*, *code injection* and *sandboxing* would be valuable additions to the field. My main contribution is a proof of concept Java implementation of a genetic programming framework capable of (1) automatically generating genetic material by reflecting upon the problem context, (2) allowing individuals to take advantage of existing data structure implementations, (3) compiling and injecting individuals into the problem codebase on-the-fly and (4) sandboxing individuals to ensure safe execution and evaluation.

Contents

1 Project synopsis **4**

2 Introduction **5**
2.1 Preface . . . 5
2.2 Illustrating the challenge behind genetic programming . . . 6
2.2.1 Search spaces . . . 6
2.2.2 Searching with genetic programming . . . 7
2.3 Terminology . . . 8
2.4 Implementation structure . . . 9
2.4.1 System architecture . . . 9
2.4.2 Source code structure . . . 10
2.4.3 Source code notes . . . 10
2.5 Example usage . . . 11

3 Reflection **12**
3.1 Specifying classes to reflect upon . . . 12
3.2 Creating the initial gene pool . . . 13
3.2.1 The problem instance . . . 13
3.2.2 Initial reflection run . . . 13
3.2.3 Results report . . . 14
3.3 Individual gene pool expansion and sticky genes . . . 15
3.4 Implementation snippet . . . 16

4 Data structures **17**
4.1 Supporting Java generics . . . 17
4.1.1 Resolving type parameters . . . 18

5 On-the-fly compilation and injection **19**
5.1 Generating source code of individuals . . . 19
5.1.1 Imports . . . 19
5.1.2 Class definition . . . 20
5.1.3 Methods . . . 20
5.2 Compiling individuals . . . 21
5.3 Code injection . . . 21

6 Execution sandboxing **22**
6.1 Termination . . . 22
6.2 Denial of service . . . 23
6.3 Malicious actions . . . 23

7 Example output **25**

References **26**

1 Project synopsis

The foundation of this project is the field of genetic programming. Genetic programming is an interesting combination of computer science and biology, in which programmers write caretaker software to create and evolve masses of automatically generated programs. Inspired by evolution, these programs are mated and mutated through numerous generations while enduring constant selection pressure by the caretaker to solve a difficult programming problem. This complex process is all put into action in the hope that, at some point, one of the programs will excel at solving the problem. Classic targets of genetic programming include optimization problems, artificial intelligence and novel algorithms. The goal of this project is to explore the possibility of introducing several modern programming concepts to the field, which is otherwise dominated by conventional mathematical methodologies.

I seek to implement a framework to explore the following concepts in relation to genetic programming:

- **Context reflection**
  Genetic programs are traditionally expected to be completely self-contained and only take advantage of custom crafted components. This limitation is problematic for problems which are not easily isolated but are part of a larger codebase and thus very context-dependent. Implementation of a framework capable of reflecting upon the problem context and reuse the codebase could significantly ease the barriers of using genetic programming to solve these problems.

- **Data structures**
  Modern programming has heavy reliance on data structures. However, these are rarely available in genetic programming frameworks unless the genetic programs evolve them on their own. Common data structures like lists, maps and sets could be introduced, and these are often available in the programming language of choice already. Leveraging these could open up an opportunity for shorter and more readable programs while taking advantage of proven and efficient ways of managing data at runtime.

- **On-the-fly compilation and injection**
  Genetic individuals are traditionally evaluated by custom evaluation functions coded into the custom crafted components. By taking advantage of on-the-fly compilation and injection, the individuals can be evaluated in the original problem codebase while relieving the programmer of tedious work.

- **Execution sandboxing**
  To support modern programming features such as data structures and loops, genetic programs need to be able to either directly or indirectly allocate memory as well as execute code that might not terminate. This implies that genetic programs can act maliciously by, among other things, consuming a critical amount of system resources. Execution sandboxing is an interesting solution to the issue that gives the genetic program free rein to consume resources and perform actions while the framework can enforce hard limits on consumption and protect the host system from damage.

2 Introduction

2.1 Preface

> True beauty is measured not by appearance but by actions and deeds. Many have eyes, but few have seen. Of all here, you saw the beauty behind the illusion. And you alone shall be blessed with My gifts.
>
> — *Scriptures of Lyssa, 45 BE*

Genejector is a genetic programming framework. Its proof of concept implementation is the core product of my computer science bachelor thesis. I encourage the reader to open up their favourite Java IDE[1] and consult the source code (which is not included in full in this report due to sheer size) during reading. The source code is available at my website: http://zero3.dk/.

This report does not contain a thorough introduction to genetic programming. It is out of scope for my project and several people have done this already. Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee provides an excellent introduction in *A field guide to genetic programming* [1]. The 10 ECTS[2] allocated for this project in my study curriculum also sets an upper limit on the number of things I can possibly cover. For these reasons, I will only briefly discuss some of the generalities about genetic programming and skip straight to my work on the specifics mentioned in my synopsis. The same is true for my proof of concept implementation which does not currently support many important language constructs such as IF, FOR and WHILE statements.

For similar reasons, I have spent no time comparing and benchmarking my framework against existing solutions nor tested it against classic genetic programming benchmark problems such as the *Santa Fe Trail* [2]. Only simple examples, mostly targeting the goals listed in my project synopsis, are included in my code. I hope to eventually get the chance to explore these areas further in the future. Especially benchmarking is an open issue within the field, without any real consensus [3]. Work is being done towards solving this though [4].

On a related note, Genejector is a clean room implementation. I only briefly tested other genetic programming frameworks while still searching for a topic for my bachelor thesis. Since my work began, I have not consulted existing solutions. I suspect that this indeliberate choice might have led to a different solution structure than commonly seen in the field. I have included a brief overview of the architecture of the framework in section 2.4.1 and the structure of the source code in section 2.4.2.

Regarding the source code: Genejector is implemented in the Java programming language. Many of the goals behind Genejector are in fact inspired by functionality included in Java and the Java virtual machine. Some (perhaps all) of these features are also available in other software development environments. The choice of Java certainly has both advantages (huge library, fast compilation, relatively easy syntax, ...) and disadvantages (boilerplate code, execution overhead, bureaucracy, ...) not to mention the inconvenience of running a Java virtual machine. Nevertheless, as clearly seen later in this report, some of the principles behind how Java works are actually huge advantages in the field of genetic programming.

Much thanks goes out to Simon M. Lucas for his paper *Exploiting Reflection in Object Oriented Genetic Programming* [5] as well as Riccardo Poli et al. for their previously mentioned work [1]. Much of my inspiration behind this project is based on their introductions to the field.

~ Christian Funder Sommerlund

[1] *Integrated Development Environment* – like NetBeans or Eclipse. Or just your favourite text editor, if you are that old-school.
[2] *European Credit Transfer and Accumulation System.* This bachelor thesis is rated at 10 ECTS which is rated at 275 hours of work (all inclusive) in Denmark.

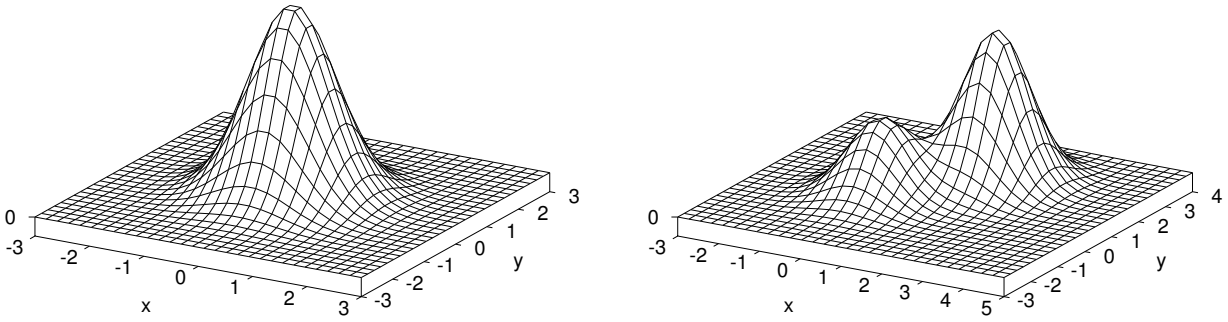
2.2 Illustrating the challenge behind genetic programming

> For I am your goddess, and I will give blessings to all who follow these teachings.
>
> — *Scriptures of Dwayna, 115 BE*

Before diving into the project specifics, I will briefly touch the topic of search space complexity for the slightly mathematically inclined. A basic understanding of the topic should be of great advantage in understanding how genetic programming works and why it is even used. Once grasped, it should also be clear that limiting the search space is of great importance to genetic programming, and thus why some of the design choices behind Genejector, such as strong typing and the inclusion of data structure libraries are very significant to its efficiency.

2.2.1 Search spaces

The 3D plot in figure 1a visualizes the possible results of an arbitrary mathematical function of two variables. As usual, the resulting value for a given input of a specific x and a specific y is visualized through the height on the z-axis at the corresponding location in the graph. The figure represents the *solution space,* or *search space* in the context of genetic programming, of the function (safely assume that the function always returns 0 outside the boundaries of the graph).



(a) An easy search space

(b) A harder search space

Figure 1: Examples of simple search spaces

Suppose that we want to find the values of x and y such that we obtain the highest possible result. One can easily observe this point to be $(0, 0)$ in figure 1a.

Now imagine that we could represent computer programs as values of x and y. Imagine, for example, that programs only consist of two commands and that the list of possible commands are numbered with integers. Then x and y becomes integers indicating which commands to put in the first and second command slot of the program respectively. Now imagine that we define the return value of our function to be a test score of the program provided as input. This test score is calculated based on how well the program performs some arbitrary task (be it adding two numbers or playing poker online). In the field of genetic programming, this test score is commonly called *fitness* and 0 is defined as being the best possible fitness. This project uses the opposite scale with 0 being the worst possible result, and simply calls it **score**.

Hold this thought while returning to figure 1a. Observe that different combinations of x and y values yield different scores (simply ignore the fact that the graph is continuous and actually represents a mathematical function). For example, $(0.5, -1)$ is a relatively poorly performing program while the best program, the *optimal solution*, is $(0, 0)$ as mentioned earlier.

The goal is thus to somehow find the values of x and y that leads to the optimal solution (or at least a pretty good solution). Obviously we do not know the location of the optimal solution in the search space beforehand.

As a matter of fact, we often do not even know the shape of the search space, commonly called the *fitness landscape*. The search space for most mathematical functions and programs are in fact infinitely long! This is easily proven by intuition: Simply consider for a second that you can always increase the value of x or y in the mathematical function example or add another command in the program example.

Let us consider a simple, yet effective, algorithm to solve problems with a fitness landscape like the one in figure 1a: *hill climbing*. As the name suggests, the idea is to keep "climbing" up a "hill" in the fitness landscape until you can climb no further and then return that solution. The algorithm simply starts at any solution in the search space and keeps changing the x and y variables in such a way that it moves to better and better positions in the fitness landscape. This is practically done by testing a number of neighbor solutions, *mutations*, and picking the best one as the next *candidate solution*. Repeat until no neighbor solutions are better than your current position and you are done.

In mathematical functions you can move from one candidate solution to another simply by incrementing or decrementing the x and y inputs by predefined constants. In programs you can, for example, change a single command into another or change the argument of a function from one value to another.

While hill climbing works very well for problems with fitness landscapes shaped like the one in figure 1a, it fails miserably for most other. Consider for example the landscape in figure 1b with two hills, one leading to only a half as good solution as the other. If the hill climbing algorithm gets a foothold on the sub-optimal hill, the algorithm will return a sub-optimal solution. Expansions of the hill climbing algorithm, such as *tabu search* and *random-restart hill climbing*, attempts to get around this issue by searching a larger neighbourhood and different parts of the search space respectively. This is a perfectly fine strategy for some kind of problems, but still relies on assumptions about the shape of the fitness landscape to be effective. Consider how hill climbing and similar methods would perform against a fitness landscape as depicted in figure 2. Suddenly these methods become much less appealing.

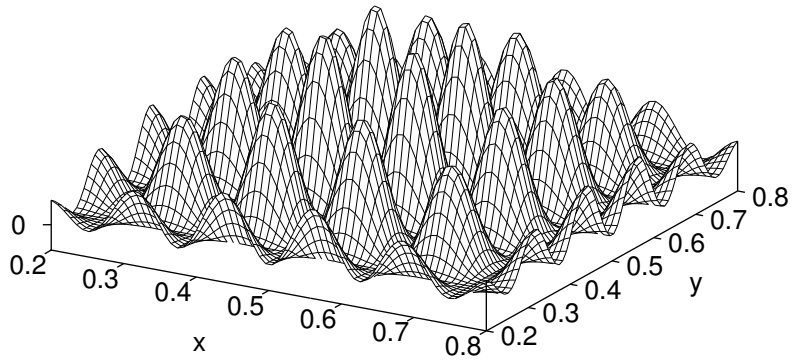


Figure 2: A much harder search space

2.2.2 Searching with genetic programming

Imagine a search space infinitely large and instead of smooth hills and valleys consists of a landscape that can abruptly change from useless flat ground to floating mountains of high performance. Add large holes in the landscape here and there (representing invalid programs) and many, many more input dimensions than the two used in our simplified example above. If it was possible to visualize the search space of programs in an intuitive way, such a landscape would be much more likely than the ones in figure 1a, 1b and 2.

Thankfully, we do not have to be able to plot programs in order to find good ones. But having the mathematical simplification above in mind can help understanding the conceptual challenge in finding them.

One of the reasons why genetic programming is such an interesting method is that it does not depend directly on the layout of the fitness landscape (even though it is often able to take advantage of it).

Consider the commonly used operator in genetic programming called *crossover*. Also inspired by biology, this operator picks two random **individuals** from the current population of programs (usually weighted by fitness), copies them and picks a random breeding position in the source code of each of them. It then throws away everything on one side of each breeding position and attaches the remaining two parts together to generate a completely new program.

In terms of the search space visualization in figure 2, imagine that the two picked individuals were located at some of the smaller hilltops. Using the crossover operator is then like performing a low-gravity BASE jump[3] from one of the smaller hilltops towards the other. Without a parachute. If you are lucky, you land on the side of another hill. If not, you fall down and die. This is what happens to the programs as well. If the newly generated genetic individual happens to perform well, the individual gets a high score and has a better chance of ”surviving” into the next generation. If the new individual is an invalid or low-scoring program, it will be most likely be ignored by the genetic programming framework instead.

The other classic genetic programming operator is *mutation*. Recall that the hill climbing algorithm mentioned earlier does several mutations per iteration, evaluates them all and picks the best. The mutation operator used in genetic programming is usually even simpler: Perform a single random mutation and put the resulting individual into the next generation, no matter how good or bad performing the omniscient **framework** believes it to be. This might move the individual further up or down a hill or it might nudge it off an edge into fitnessless void.

It should be intuitively clear that it is of great advantage for a genetic programming framework to be designed in such a way that it minimizes the size of the low performing and invalid areas in the fitness landscape. The more fitness in the landscape, the better chances of generating good programs in reasonable time.

2.3 Terminology

For the sake of consistency, I have attempted to reuse as much terminology from existing literature as possible and only invented new terms for concepts that I did not find any equivalent existing terms for. I apologize beforehand for any confusion caused by term redundancy.

Java	The programming language.
JVM	A **Java** virtual machine in which **Java** programs are executed.
framework	A genetic programming software framework, like Genejector.
project	A programming project to which Genejector is attached.
problem	A task inside a **project** to which a solution needs to be found by the **framework**.
individual	A genetically programmed candidate solution to the **problem**. This term is used for both the internal data structure representation and compiled code representation. The code mostly uses the terms *mortal* and *risen* respectively, to separate the two versions.
problem class	A **Java** class modeling a **problem**.
problem object	An instance of a **problem class** provided to **individuals** upon execution inside a **project**.
gene	The lowest level component of an **individual**. Genes are attached to each other in a tree structure with root in the **individual**.

[3] *buildings, antennas, spans and earth.* BASE jumping is an activity in which you make free falls from high positions using a parachute to save your life. The most dangerous ”recreational activity” in the world, according to Wikipedia.

score A non-negative integer rating the ability of an **individual** to solve a **problem**. In Genejector, a **project** is responsible for measuring this and report it back to the **framework**. Almost equivalent to the term *fitness* commonly used in the field, except that the scale is inversed: A *fitness* of 0 is given for the *best* possible performance while a score of 0 is given for the *worst* possible performance.

sticky gene A **gene** referencing another **gene** that, when the other **gene** is removed from the **individual**, automatically references another compatible **gene**.

run A single execution of a **framework** towards solving a **problem**. Some **frameworks** support performing more than one **run** per execution.

2.4 Implementation structure

2.4.1 System architecture

The architecture of Genejector spans no less than three **JVM** instances, as roughly illustrated in figure 3. These are named the *project JVM*, *realm JVM* and *risen JVM*.

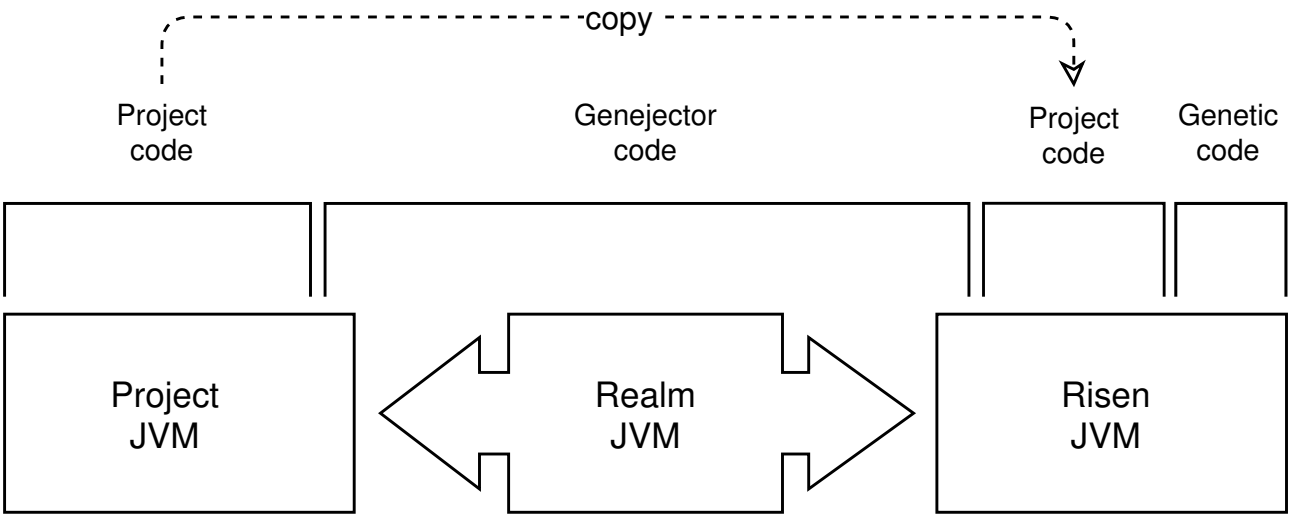


Figure 3: Genejector architecture overview

The project JVM is where it all starts. This is the JVM launched by the user to run his **project**. The user has added the Genejector .jar library to the project, and calls the static method GENEJECTOR.GENEJECT() (located in the package GENEJECTOR.SHARED) at some point in the project code where Genejector is to be used. The project is responsible for having changed the default settings as needed before doing so, using the various methods on the SETTINGS object (also from GENEJECTOR.SHARED) obtainable via the static method GENEJECTOR.GETSETTINGS(). Once called using GENEJECTOR.GENEJECT(), Genejector will escape out of the project JVM by spawning the realm JVM for itself. This is done to isolate the rather resource intensive breeding process from the project JVM and in general affect it as little as possible.

When the realm JVM has started, it will request the settings object from the project JVM using **Java** sockets and object streams. The project JVM is then paused (the thread which called GENEJECTOR.GENEJECT(), that is) for the duration of the **run**. Genejector then starts working on the **problem** by first reflecting through the project code (as further described in section 3) to assemble the initial **gene** pool. Evolutionary breeding commences, and continues until some termination criterion is met, as specified in the SETTINGS object. The best **individual** found at termination (if any) is returned to the project JVM which is then resumed upon termination of the realm JVM.

The last JVM, the risen JVM, is used by the realm JVM for scoring the individuals. For security and stability reasons elaborated in section 6, we cannot safely execute genetic code in the same process as the framework itself, and thus delegate this task to the risen JVM which runs a copy of the original project (by simply copying and reusing the command line used to launch the project JVM). The risen JVM itself is attempted reused for scoring multiple individuals to avoid unnecessary overhead, but should it die during the process it is simply respawned.

2.4.2 Source code structure

The source code is organized in Java packages reflecting the overall system architecture. The code involving each of the three JVMs are thus kept separate except for a few exceptions like shared utility classes.

The following list gives a quick overview of the packages (all inside the GENEJECTOR parent package):

EXAMPLES	A couple of "Hello World!" example projects. These are small proof of concept **projects** targeting various features of Genejector.
PROJECT	Various classes for the project JVM.
REALM	Various classes for the realm JVM.
REALM.BREEDINGOPERATORS	Breeding operators. Included are standard mutation and statement adding/removing operators.
REALM.GENES	Prototype **genes**. Instances of these are added to the gene pool and later cloned into the individuals during breeding.
REALM.GENETRAITS	Interfaces implemented by the various genes according to their functionality.
REALM.GODS	The core functionality of Genejector. These classes take care of the evolutionary process (DWAYNA), reflection (KORMIR), growing genes (ABADDON), compilation (BALTHAZAR), scoring (MELANDRU) and statistics (LYSSA).
REALM.MORTAL	Classes capable of representing an individual in a tree structure and the transformation into Java source code.
RISEN	Various classes for the risen JVM.
SHARED	Various classes shared between the JVMs. Most notably GENEJECTOR and SETTINGS.
SHARED.DATASTRUCTURES	Various custom data structures.
SHARED.EXCEPTIONS	Exceptions and exception wrappers thrown throughout Genejector.
SHARED.MESSAGES	Message containers for messages sent between the JVMs using sockets.
SHARED.UTIL	Various utilities.

2.4.3 Source code notes

During my work on Genejector I placed many comments and notes throughout the code. Many of these document possible extension points (marked "TODO") and would-be-nice-to-have features. I have deliberately left these intact as I believe they provide the reader with valuable documentation of the thoughts behind the design of the code and what ideas I have for it for the future.

Code listings throughout the report are marked with file paths and ID tags. Corresponding start and end tags are to be found in the source code of these files.

2.5 Example usage

Figure 4 shows one of the included "Hello World!" example projects, TripleCopyExample. Example output of a run against this example is presented in section 7 for the curious reader.

.../examples/TripleCopyExample.java [L7]

```
7  public class TripleCopyExample
8  {
9    public static void main(String[] args)
10   {
11     TripleCopyProblem problem = new TripleCopyProblem();
12
13     // 1) Change default settings as needed
14     Genejector.getSettings().setScoreLimit(6);
15     Genejector.getSettings().setPrintAllIndividuals(true);
16     Genejector.getSettings().setAddDefaultClasses(false);
17     Genejector.getSettings().addClass("java.lang.Integer", false);
18
19     while (!Genejector.isSolutionFound())
20     {
21       // 2) Request a new candidate solution
22       Genejector.geneject(problem.getClass());
23
24       // 3) Test candidate solution
25       long score = 0;
26       problem.setInputs(4, 5, 6);
27       Genejector.execute(problem);
28       score += ((problem.getOutput1() != null && problem.getOutput1().equals(4)) ? 1 : 0);
29       score += ((problem.getOutput2() != null && problem.getOutput2().equals(5)) ? 1 : 0);
30       score += ((problem.getOutput3() != null && problem.getOutput3().equals(6)) ? 1 : 0);
31
32       problem.setInputs(-1337, 42, 27);
33       Genejector.execute(problem);
34       score += ((problem.getOutput1() != null && problem.getOutput1().equals(-1337)) ? 1 : 0);
35       score += ((problem.getOutput2() != null && problem.getOutput2().equals(42)) ? 1 : 0);
36       score += ((problem.getOutput3() != null && problem.getOutput3().equals(27)) ? 1 : 0);
37
38       // 4) Send back score
39       Genejector.submitScore(score);
40     }
41
42     System.out.println("Solution: " + Genejector.getSolutionSourceCode());
43   }
44 }
```

.../examples/TripleCopyProblem.java [L2]

```
8    // These 3 inputs ...
9    private Integer input1 = null;
10   private Integer input2 = null;
11   private Integer input3 = null;
12
13   // ... should be copied into these 3 outputs
14   private Integer output1 = null;
15   private Integer output2 = null;
16   private Integer output3 = null;
17
18   public void setInputs(Integer input1, Integer input2, Integer input3)
19   {
20     this.input1 = input1;
21     this.input2 = input2;
22     this.input3 = input3;
23   }
24
25   @Override
26   public void resetState()
27   {
28     output1 = null;
29     output2 = null;
30     output3 = null;
31   }
```

Figure 4: TripleCopyExample, in which the task is to copy integers from three input variables to three output variables (package declarations, imports and standard Java getters and setters not shown).

3 Reflection

> Yet the power of a god cannot be destroyed, and Kormir, making a choice that only a mortal could make, did take upon herself the mantle of the Goddess of Truth, with all its power and responsibility, all its dominion and duties.
>
> — *Scriptures of Kormir, 1075 AE*

Genetic programs are still programs, and thus, like most other programs, are built from source code. In the classic parse tree model [6], which Genejector builds upon, each node is responsible for generating source code representing its own functionality, without necessarily knowing anything about the rest of the tree (not even its own parent or children nodes). A simple mathematical addition node might for example just output the character '+' prepended and appended with the source code of its first and second child respectively. These might be leaf nodes returning integers like 13 and 37, or might be functions with child nodes of their own.

Obviously these nodes (genes in this project) need to be made available to the framework in the first place somehow. Frameworks typically ship with a number of pre-programmed general-purpose nodes such as common mathematical functions. If the user of the framework wants the individuals to take advantage of functionality from their own project, they need to develop nodes for each piece of functionality themselves. Needless to say, this can quickly become tiresome, and the genetic material available to the individuals becomes heavily biased by what the programmer believes is needed to solve the problem. As a programmer typically will use genetic programming to solve (or optimize) a problem he cannot find good solutions to himself, this leads to an interesting conflict: How can the programmer know which functionality is required to solve the problem then?

Context-dependent functionality aside, many modern programming languages (Java, Python and PHP to name a few) also have an enormous library of functionality available out of the box. These features (especially data structures, which are covered further in section 4) are commonly used to solve problems by programmers, yet typically also unavailable to genetic programs unless encoded into genetic material by the programmer himself.

Genejector aims to solve both of these similar issues by taking advantage of reflection not only against the project code but also against the Java library itself. The idea of taking advantage of reflection is not new [5], but unfortunately still not actively used. Neither of the two major genetic programming frameworks for Java, ECJ [7] and JGAP [8], supports it. A single effort to introduce reflection to genetic programming in Java, GenPro [9], was started in 2009 but abandoned again same year without ever reaching a mature state.

3.1 Specifying classes to reflect upon

Genejector is capable of reflecting upon Java classes and recursively generate genetic material from class members for use in the breeding process. While it is theoretically possible to encode the whole Java library as well as the project code as genetic material, doing so would be counter-productive and can even be dangerous for the user (more on this in section 6.3 about malicious actions). Even though a wide selection of genetic material can be an advantage, as previously discussed, one has to keep in mind that for each gene added to the gene pool adds additional complexity to the already very difficult search space. For these reasons, Genejector only reflects upon classes explicitly whitelisted by the user. A built-in selection of core Java classes (String and Integer at the moment) are added by default, but the user may specify additional classes directly from the project code as shown in figure 5. The last argument of the addClass() method indicates whether individuals are allowed to create new instances of the class. If not, they will only manipulate existing instances they can get a hold of references to. Individuals bred in the CrazyExample project are thus allowed to use both existing CrazyKeyring and CrazySafe objects but are also allowed to directly instantiate new CrazySafe objects on their own initiative.

.../examples/CrazyExample.java [L1]

```
18      Genejector.getSettings().addClass("genejector.examples.CrazyKeyring", false);
19      Genejector.getSettings().addClass("genejector.examples.CrazySafe", true);
```

Figure 5: Whitelisting classes for reflection

The previously shown example in figure 4 disables the addition of the default classes (using SETTINGS#SETADDDEFAULTCLASSES(FALSE)) and adds only JAVA.LANG.INTEGER to limit the search space as there is no need for strings and the like in that example.

From Genejector's perspective there is no difference between classes originating from the Java library and those originating from project code. They are treated equally.

3.2 Creating the initial gene pool

All genetic individuals share the same starting point inside the project when they are to be executed. This is the point where the project calls the static GENEJECTOR.EXECUTE() method and execution control is handed over to the individual. This starting point never changes, and thus never does the set of objects, fields and methods available at this point. The initial gene pool based on these are created upon startup and reused throughout the **run**.

3.2.1 The problem instance

Once genetic individuals are called inside the project context at runtime, they expect a reference to an existing object inside the project. This is called the **problem object** and is an instance of the **problem class**. This class is designed by the user and is expected to hold fields and methods providing the individual with all the things required to interact with the project. Depending on the problem, this object could contain various things. For example, the TRIPLECOPYPROBLEM from figure 4 contains three input variables, three output variables and standard Java getters and setters.

Genejector is also capable of using static methods and fields (described further in section 3.2.2), but most problem contexts should involve at least a few references to existing object instances.

To facilitate the handing over of the problem object to the individual, all individuals implement expect the object as an argument for their EXECUTE() method. The individual obviously must know the class of this object beforehand (as Java is a statically typed language), so it is specified as an argument to Genejector in the initial call to GENEJECTOR.GENEJECT() and then used when breeding the individuals.

3.2.2 Initial reflection run

As mentioned above, Genejector will start out by performing an initial reflection run in order to assemble the initial gene pool. The pool starts with just a single gene: A USEFIELDGENE providing a reference to the problem object provided to the individual as described in section 3.2.1.

One of the principles behind my implementation of reflection is that genes representing non-static fields and methods of a class are not added to the gene pool before an instance of the class exists. For example: A gene representing the LENGTH() method of a STRING will not be made available in the gene pool before at least one STRING reference is available either directly or indirectly (perhaps as the return value of another method). This is done to greatly reduce the search space by eliminating a lot of useless mutations.

In other words: The availability of a gene in the gene pool depends on the availability of other genes in the gene pool. Consider a gene pool which initially have no String genes available but during reflection gains one MethodGene that returns a String. The addition of that gene means that String objects are now available, and String-dependent genes should thus be added to the pool as well. In order to handle these dependencies, Genejector will repeatedly reflect upon the problem until no more genes are added to the gene pool.

The actual reflection is done per whitelisted class. If an instance of a given class is available, all public methods and fields are added as genetic material to the gene pool. Static fields, methods and constructors (if instantiation is allowed for the given class) are always added (as they are always available, even without a class instance) if their dependencies are met. Keep in mind though, that a static member can make an instance of a class available and thus allow the reflection run to escape static context.

Note that as the initial gene pool was seeded with the problem object, all methods and fields from the problem class will be added to the initial gene pool during the initial reflection run (once again assuming that their dependencies are met, of course).

3.2.3 Results report

Once the initial gene pool has been assembled, a full reflection report is printed to standard output. The report contains the selected classes, the genetic material extracted from them as well as any classes discovered underway that were ignored. Figure 6 shows one such report.

```
 | Approved breeding classes:
 | | Without instantiation:
 | | | genejector.examples.TripleCopyProblem
 | | | java.lang.Integer
Kormir: Created initial gene pool:
 | <Instruction>:
 | |   [G3] NewVariableGene [java.lang.Integer]
 | |   [G6] MethodGene      [genejector.examples.TripleCopyProblem#setOutput1(java.lang.Integer)]
 | |   [G9] MethodGene      [genejector.examples.TripleCopyProblem#setOutput2(java.lang.Integer)]
 | | [G12] MethodGene       [genejector.examples.TripleCopyProblem#setOutput3(java.lang.Integer)]
 | | [G13] MethodGene       [java.lang.Integer#toString()]
 | | [G14] MethodGene       [java.lang.Integer#hashCode()]
 | | [G15] MethodGene       [java.lang.Integer#compareTo(java.lang.Integer)]
 | | [G16] MethodGene       [java.lang.Integer#byteValue()]
 | | [G17] MethodGene       [java.lang.Integer#shortValue()]
 | | [G18] MethodGene       [java.lang.Integer#intValue()]
 | | [G19] MethodGene       [java.lang.Integer#longValue()]
 | | [G20] MethodGene       [java.lang.Integer#floatValue()]
 | | [G21] MethodGene       [java.lang.Integer#doubleValue()]
 | | [G22] MethodGene       [genejector.examples.TripleCopyProblem#setInputs(java.lang.Integer, java.lang↩
   .Integer, java.lang.Integer)]
 | <Statement>:
 | |   [G4] InstructionGene []
 | genejector.examples.TripleCopyProblem:
 | |   [G1] UseFieldGene    [problem]
 | java.lang.Integer:
 | |   [G2] MethodGene      [genejector.examples.TripleCopyProblem#getInput1()]
 | |   [G5] MethodGene      [genejector.examples.TripleCopyProblem#getOutput1()]
 | |   [G7] MethodGene      [genejector.examples.TripleCopyProblem#getInput2()]
 | |   [G8] MethodGene      [genejector.examples.TripleCopyProblem#getOutput2()]
 | | [G10] MethodGene       [genejector.examples.TripleCopyProblem#getInput3()]
 | | [G11] MethodGene       [genejector.examples.TripleCopyProblem#getOutput3()]
Kormir: Skipped all fields, methods and return values involving these non-approved classes:
 | boolean
 | byte
 | double
 | float
 | int
 | java.lang.Class
 | java.lang.Object
 | java.lang.String
 | long
 | short
```

Figure 6: Reflection report from Genejector attached to TripleCopyProblem

This report is very important, as it provides the user with a significant insight into the initial search space which the run will be based on. If any class is not whitelisted under the *approved breeding classes* section, genetic material for it will not be available in the gene pool. If a class is listed under the *skipped* section, it means that Genejector found a reference to the class through the problem context but ignored it because it was not on the whitelist. The main section of the report contains a human readable version of the initial gene pool. These are subsectioned by the type of parent to which they are attachable (their "return type"). A single gene line contains the gene's prototype ID (a simple incrementing number prefixed with a 'G'), the type of the gene (its Java class name) as well as a details section in square brackets with information depending on the type of gene, typically specifying the type of childs it can attach and/or the name of the field or method it represents.

3.3 Individual gene pool expansion and sticky genes

As noted earlier, Genejector limits the search space by only including genes in the gene pool when their dependencies are expected to be satisfiable. This implies that we need to add such genes during the breeding process as references become available.

This is done when growing new genes using GENEJECTOR.REALM.GODS.ABADDON#GROWCHILD(). Before actually picking and attaching a gene at the requested point in an individual, Abaddon asks Kormir to reflect upon the individual using GENEJECTOR.REALM.GODS.KORMIR#REFLECTMORTAL() and add any new genes to a copy of the initial gene pool. Kormir performs a bottom-to-top internal reflection on the part of the gene tree of the individual *preceding* the breeding point. During this reflection, Kormir looks for genes of the type NEWVARIABLEGENE within a scope available at the breeding point. These genes, as the name suggests, represents the creation of a new variable. Kormir looks upon the type of the variable (called *TypeOfVariable* from now on) and makes three additions to the gene pool:

- the NEWVARIABLEGENE itself, under a special variable tag: "<Variable: *TypeOfVariable*>".
- a USEVARIABLEGENE of type *TypeOfVariable*. This gene is capable of being attached to parents later on in the tree which requires childs of type *TypeOfVariable*. This gene is hollow and its only purpose is to hold an internal reference to a "<Variable: *TypeOfVariable*>"-typed gene (as the one just added) in order to be able to obtain the name of the variable when generating the source code of the individual.
- an ASSIGNMENTGENE capable of assigning a new object to the variable.

The intermediary USEVARIABLEGENE gene is put into place to prevent also assigning the NEWVARIABLEGENE as a child to a gene later in the tree, as this will violate the topology contract of the tree structure and significantly complicate other parts of the framework.

This trick paves the way for a concept I have chosen to call **sticky genes**: As the *consumers* of variables are now separated from the *producers* of variables, only linked together by a type specification, we can dynamically reassign different producers to different consumers. This becomes immensely useful when using the crossover operator (and others – even mutation) on individuals. Imagine that the crossover point is selected to lie between the creation of a variable and one of its consumers later on the tree (which is quite likely to happen). When the subtree with the variable consumer is combined with the subtree of another individual, the gene become invalid – the variable it was using just disappeared!

There are various ways to handle these broken consumer genes, but many of the easy ones have serious limitations. One way is simply letting the individual die "naturally" when it fails to compile because of an undeclared variable reference. Another way is to automatically remove all consumer genes (and their children subtrees)

when their variable dependencies disappear. But both of these solutions are quite counter-productive and effectively makes variables largely incompatible with especially the crossover operator. A third way is to name variables with consecutive numbers (prefixed with one or more letters in the case of languages like Java where numerical variable names are not allowed), and let the consumers use another variable with the same name from the new subtree (if one actually becomes available, that is). The main problem with this method is that it is not type safe. There is no guarantee that the new variable with the same name is also of the same type as the previous variable of that name (it may in fact be quite unlikely).

This is where the UseVariableGene sticky gene earn its use. Sticky genes are named so because they "stick" to one variable until it is no longer available (because of a crossover operation, for example). When that happens, they "restick" to another variable of a compatible type (chosen at random in Genejector). In other words: Individuals can gracefully recover from variable producers and consumers being torn apart as long as at least one compatible producer is present in the new individual of the consumers.

3.4 Implementation snippet

The reflection engine in Genejector, Kormir, is one of the most complicated parts of the project. Nevertheless, it is worth showing a small part of it to get an idea of how it works. The three things reflected upon are fields, methods and constructors. Figure 7 shows the code responsible for reflecting upon constructors of a class.

.../realm/gods/Kormir.java [L3]

```
307     if (whitelistedClasses.get(rawClassName).instantiable)
308     {
309       constructorLoop:
310       for (Constructor<?> constructor : clazz.getConstructors())
311       {
312         List<Type> argTypes = new LinkedList<Type>();
313
314         for (java.lang.reflect.Type javaArgType : constructor.getGenericParameterTypes())
315         {
316           try
317           {
318             // Resolve generics and type parameters
319             Type argType = resolveJavaType(clazz, classParameterTypes, javaArgType);
320
321             // Ensure availability in the gene pool
322             if (genePool.get(argType) == null || genePool.get(argType).isEmpty())
323             {
324               continue constructorLoop;
325             }
326
327             argTypes.add(argType);
328           }
329           catch (NonWhitelistedTypeException ex)
330           {
331             // No go. Log and skip this constructor.
332             skippedClasses.add(ex.getOffendingType());
333             continue constructorLoop;
334           }
335         }
336
337         // All good. Infuse genes into gene pool.
338         Type newType = Type.getType(whitelistedClasses.get(rawClassName).className);
339         infuseNewObjectGene(genePool, newType, argTypes);
340         infuseNewVariableGene(genePool, newType);
341       }
342     }
```

Figure 7: Constructor reflection

As seen on the first line, this section of the code is only executed if the class is specified as instantiable (as shown in figure 5). If it is, each constructor is considered for inclusion in the gene pool in the outer for loop. The inner for loop resolves the type of each constructor argument, verifies that it is whitelisted and is actually available

in the gene pool (if not, we do not add the constructor at this time for reasons previously discussed). If all is good, a NewObjectGene and a NewVariableGene is added to the gene pool. First mentioned contains the code for the Java new statement which calls the constructor. Last mentioned makes it possible to create new variables of the type of the corresponding type. As the reader might have guessed, the gene pool is actually a Set which does not allow duplicate elements. Two additions of the same kind of NewVariableGene thus do not actually result in double presence in the gene pool. This means that we can safely re-reflect upon a class when we suspect it might give new genetic material – without messing with the balance of the gene pool.

4 Data structures

> Abaddon! Lord of the Everlasting Depths, Keeper of Secrets, open mine eyes and bestow upon me the knowledge of the Abyss that I might smite mine enemies and send them to the watery depths!
>
> — *Scriptures of Abaddon, 1 BE*

Data structures have been related to genetic programming before. Probably most famously, W. B. Langdon showed that it is possible for genetic programs to evolve data structures on their own [10].

I have taken a different approach to the topic. I see commonly used data structures such as lists, sets and maps to be important *tools* for genetic individuals to take advantage of, but not *goals* of the genetic programming itself. Evolving data structures during a run seems redundant unless the goal is to find new or better versions of them, but in most cases that will not be the purpose. Excellent implementations are nowadays available out of the box in many modern programming languages anyway – Java included.

Genejector integrates the existing data structure implementations in the Java Collections framework into the genetic programming process instead. This is done by reflecting upon the relevant Java classes such as java.util.LinkedList and java.util.HashMap and adding corresponding genetic material to the gene pool. Recall that classes from the Java library are reflected upon in the exact same way as project specific classes (discussed in section 3.1). The method of specifying classes to be reflected upon (as shown in figure 5) is therefore still the same.

4.1 Supporting Java generics

There is one catch with supporting the built-in Java collections though: Java generics. In order to assure type safety at compile time, the user must specify, at instantiation, which kind of elements a collection should be able to hold. Without, the compiler cannot assure that the element you pull out of your KennelOccupants collection at runtime is actually a Dog object and not a NaughtyCat object. Hence Java introduced generics in Java 5, requiring *parameterization* of most collections (though still allowing the usage of the raw versions for compatibility reasons). An ArrayList containing Integer elements thus became ArrayList<Integer>.

This complicates things considerably, as an ArrayList is no longer just an ArrayList. To complicate things further, Java utilizes *type erasure* (JLS 4.6 [11]) for generic classes at compile time. This means that we will not be able to obtain the parameterization of an ArrayList or other generic objects at runtime. Thankfully, we can still obtain the parameterization of fields and arguments/return values of methods and constructors with reflection. Since Genejector only works with public fields, methods and constructors, the type erasure has no negative side-effects on this project.

4.1.1 Resolving type parameters

Supporting Java generics is still not entirely trivial, though. The easy case is handling fields, arguments and return values where classes already have been assigned to the type parameters (for example a return value of type ArrayList<Integer>). These classes are simply resolved to their *fully qualified name*, and attached to the type of the class they are parameterizing using the same syntax used in Java source code. An ArrayList containing elements of type Integer is thus translated to java.util.ArrayList<java.lang.Integer> during reflection and is then treated as any other type without parameterization. Figure 8 shows how this case is handled in genejector.realm.gods.Kormir.

.../realm/gods/Kormir.java [L9]

```
498     else if (javaType instanceof ParameterizedType)
499     {
500       // This type has at least one named parameter, like ArrayList<Integer>.
501       ParameterizedType paraTypes = (ParameterizedType) javaType;
502       String rawClass = ((Class) paraTypes.getRawType()).getCanonicalName();
503
504       // Ensure that the parameterized class is whitelisted
505       if (!whitelistedClasses.containsKey(rawClass))
506       {
507         throw new NonWhitelistedTypeException(rawClass);
508       }
509
510       List<String> paraStrings = new LinkedList<String>();
511
512       for (java.lang.reflect.Type paraType : paraTypes.getActualTypeArguments())
513       {
514         // We do not (yet) support nesting, so this will work for ArrayList<Integer>
515         // but not for ArrayList<ArrayList<Integer>> for example
516         String paraTypeName = ((Class) paraType).getCanonicalName();
517
518         // Ensure that the parameter class is whitelisted
519         if (!whitelistedClasses.containsKey(paraTypeName))
520         {
521           throw new NonWhitelistedTypeException(paraTypeName);
522         }
523
524         paraStrings.add(paraTypeName);
525       }
526
527       // Compile final type string
528       return Type.getType(rawClass + "<" + Tools.implode(paraStrings, ",") + ">");
529     }
```

Figure 8: Resolving type parameters bound to classes

As noted in the source code, this implementation is very simple and does not support nested generics or advanced features like type parameter bounds (JLS 4.5.1 [11]). It does, however, provide basic support suitable as a proof of concept.

The hard case is handling generic classes without type parameter assignments which instead contain one or more *type variables*. This occurs when we reflect upon a generic class itself (and not some instance of it). Example: java.util.HashMap is actually declared with two type variables as HashMap<K, V>. Genejector handles reflection of these classes by permutating through all possible assignments of whitelisted classes to type variables. During each permutation the class is reflected upon with the type variables locked to these assignments. Recall from section 3.2.2 that we only actually add genetic material for non-static class members if we have at least one instance of the class available (directly or indirectly). For example: The HashMap<Integer, String>#get() method is only added to the gene pool if the individual has access to at least one HashMap<Integer, String> object. Thus, while extensive, the permutation approach is an easy solution to an otherwise complicated process of resolving dependencies.

The included RotateListExample shows how individuals can learn to remove the first element of a list and reinsert it into the back of the list only by reflecting upon the problem and the ArrayList class.

5 On-the-fly compilation and injection

> Lift up thy weapons. For you are my soldiers, and must you be steadfast, strong, and brave of heart. They who neither hesitate nor stumble shall be rewarded. Then shall you have glory. Then shall your deeds be remembered for eternity.
>
> — *Scriptures of Balthazar, 48 BE*

As most Java software, Genejector is distributed pre-compiled and ready to be loaded and linked by the JVM. The individuals, however, are not. This is obviously not possible, as their source code depends on whatever project the user attaches Genejector to. Thankfully, Java supports dynamic class loading, making it possible for Genejector to write the source code, compile it in memory and inject it into the project at runtime.

5.1 Generating source code of individuals

In order to compile an individual, we first need to generate its source code. The following subsections explain the approach used in Genejector.

5.1.1 Imports

To identify a class in Java, one can either specify the fully qualified class name (FQN) upon each reference or import it once, in the imports section of the .java file, and use its simple name for the rest of the file. See figure 9 and 10 for examples.

```
1 public void getNewInstance()
2 {
3   myproject.mypackage.MyClass myObject = new myproject.mypackage.MyClass();
4   return myObject;
5 }
```

Figure 9: Example of a method using a fully qualified class name

```
1 import myproject.mypackage.MyClass;
2
3 public void getNewInstance()
4 {
5   MyClass myObject = new MyClass();
6   return myObject;
7 }
```

Figure 10: Example of a method importing a class and using its simple class name

Importing a class has the implication of mapping its simple name to its FQN. This creates potential for conflicts as FQNs are assumed to be unique while simple names are not. It is not uncommon for a project to have several packages with identically named classes (think of the many possible uses of names such as *node*, *edge*, *container* and *block*). Importing more than one of these at the same time would be an error, as there is no way to distinguish their identical simple names from each other.

To avoid the bookkeeping required to prevent overlapping simple names, Genejector only produces code using FQNs. This does produce less readable code, but since almost all generated code never will be read by a human anyway, this seems like a sensible tradeoff. The user is free to refactor the code of select individuals manually or with an IDE as needed.

This decision leads to a simple import section of generated individuals. In fact, the only import ever used is the marker interface GenejectedMortal. This interface is empty and serves no other purpose than providing readability in the Genejector source code when handling compiled individuals.

5.1.2 Class definition

As all Java code has to reside in classes, the genetic code is placed in a class named GenejectorMortal. This class serves no other purpose than being a container for the execute() method and implementing the GenejectedMortal marker interface mentioned in section 5.1.1. Note that this class name is reused by all individuals.

5.1.3 Methods

Generated individuals only contain a single method, named execute(). This is the method called by the framework to hand over execution to the individual. Everything up until now has been boilerplate code. The content of this method is the actual code of the individual.

To piece together the source code of a genetic individual, Genejector asks the root node for its source code using the getCode() method declared in the GeneTrait interface that all genes must implement. The root node recursively asks its children for their source code, appends it with any source code it might be responsible for itself, and returns the final result. Figures 11 and 12 show examples of getCode() implementations by genes.

.../realm/genes/AssignmentGene.java [L4]

```
53  public String getCode(SourceCompositionTask task)
54  {
55    // Concise: "<variable / field name>=<code of child>"
56    return reference.getCode(task) + "=" + child.getCode(task);
57  }
```

Figure 11: getCode() implementation by AssignmentGene

.../realm/genes/NewObjectGene.java [L5]

```
78  public String getCode(SourceCompositionTask task)
79  {
80    // Concise: "new <variable type>(<code of child1>,<code of child2>,...)"
81    StringBuilder string = new StringBuilder();
82
83    string.append("new ").append(variableType).append("(");
84
85    for (int i = 0; i < childGenes.size(); i++)
86    {
87      if (i > 0)
88      {
89        string.append(",");
90      }
91
92      string.append(childGenes.get(i).getCode(task));
93    }
94
95    string.append(")");
96
97    return string.toString();
98  }
```

Figure 12: getCode() implementation by NewObjectGene

5.2 Compiling individuals

With the source code ready, all there is left to do is compiling it. This could be done in old-fashioned way by saving the source code to disk, launching the *javac* Java compiler with the right arguments and reading the compiled class file back in. This, however, involves a lot of overhead and is cumbersome of nature. Instead, Genejector uses the JavaCompiler interface provided since Java 6, allowing Java applications to compile Java code themselves.

To use this technique, one must first obtain a JavaCompiler reference to the system compiler using the static method javax.tools.ToolProvider.getSystemJavaCompiler(). If the system has the required dependencies available (namely the Java Development Kit (JDK)), an object implementing said interface is returned (otherwise the method returns null).

But even with a JavaCompiler on hand, there is an obstruction to handle. By default, source code and compiled classes are read from and saved to disk respectively. Contrary to what one might believe, it is rather difficult to change this behaviour. It involves replacing the default JavaFileManager used by the compiler into a memory-based version capable of reading and writing the files directly from/to memory. A support class, MemoryJavaFileManager, written by Sun Microsystems back in 2006 for a different purpose, has been included to solve this problem. The details of how it works are neither interesting nor relevant for the project and is left for the reader to explore on his own if curious (it is located in the genejector.shared.util package).

After calling the getTask().call() method chain on the JavaCompiler object, the compiled class is retrieved through the custom memory file manager in the form of raw bytes, ready to be injected into the project codebase.

5.3 Code injection

Once an individual has been compiled, it needs to be executed inside the project context in order to let the project evaluate the individual and report its score back to Genejector. As Java loads compiled classes on the run as they are needed, it is possible to dynamically load classes into a running Java program. These dynamically loaded classes are treated just as pre-compiled and pre-packaged classes and are thus able to directly interact with the original project code. In other words, the injected code becomes part of the original project code from the perspective of the project.

When Genejector launches an instance of the project in the risen JVM, it does not run the main method of the original project immediately. Instead, it starts executing in the custom main method in genejector.risen.RisenInstanceManager responsible for opening up a communication channel to the realm JVM. Once the channel is open, the original main method is called through reflection.

The project then calls Genejector.geneject() every time it is ready to evaluate a new individual. When called, an individual is requested through the communication channel and the realm JVM sends back an individual in the form of a class in raw byte form.

Once received, the class needs to be loaded and linked into the project. There is, however, one problem with dynamically loading the classes in Java. According to the Java Language Specification, Third Edition, section 12.2.1 [11], class loaders may cache classes at their own will. Unfortunately for our case, this is done by the default Java class loader (at least in Java 6). This means that even though the implementation of our GenejectedMortal class constantly changes, the default classloader will keep reloading the first version.

The solution is simple and crude: Each time a new version of GenejectedMortal is to be loaded, a new classloader is created. This is a waste of resources, but it does not appear to be possible to circumvent the

class caching as this appears to happen in the ClassLoader superclass out of reach of the DynamicByteClassLoader subclass used to load the raw bytes. Figure 13 shows the few lines of code required to load an individual and prepare it for execution once the prerequisites are taken care of.

.../shared/Genejector.java [L6]

```
69      DynamicByteClassLoader mortalClassLoader = new DynamicByteClassLoader("GenejectorMortal", ↩
          mortalBytes);
70      Class<?> genejectedClass = Class.forName("GenejectorMortal", true, mortalClassLoader);
71      genejectedMortal = (GenejectedMortal) genejectedClass.newInstance();
72      executeMethod = genejectedClass.getMethod("execute", problem);
```

Figure 13: Dynamically loading individuals into the project context

6 Execution sandboxing

> “I am Melandru, the Mother of earth and nature. Henceforth I bind ye to these lands. When they suffer, so shall ye suffer.”
>
> — *Scriptures of Melandru, 48 BE*

There are a number of reasons why sandboxing is of great relevance to genetic programming. The following sections explain the main motivation behind the sandboxing effort in Genejector where all genetic individuals are executed inside a separate JVM instance, the risen JVM, as previously explained in section 2.4.1.

6.1 Termination

Consider for a second: What is to stop a genetic individual from looping infinitely, neither succeeding nor failing to solve the problem at hand? And even more importantly: What can one do to detect such a situation? The general answer to both questions is *nothing*. This is the *halting problem*, proved *undecidable* by Alan Turing in 1936 [12]. Practically, however, there are some things that can be done to mitigate this issue.

Either way, in order to be able to terminate stuck individuals, the framework cannot share its processing thread with the individuals. If it does, it cannot terminate anything, being asleep and all, waiting for the individual to give up control of the CPU. Which is not going to happen if it is stuck.

Even if the framework launches the individuals in separate threads or processes (something not all genetic programming frameworks currently do!), we have still not solved the halting problem. We probably cannot, but we can cheat. Simply put, if an individual does not finish within a predefined time limit it is terminated and eradicated from the population by assigning it the worst possible score.

This is the holistic approach taken by Genejector. While it is clean, predictable and easy to implement, the problem with this approach is as obvious as the approach itself: Since we terminate the individual without knowing if it is actually stuck or not, we will never know if the individual would have correctly solved the problem or not, given sufficient time.

There are other ways to circumvent the halting problem, each with their own trade-offs. It is, for example, possible to inject arbitrary stop conditions into all looping constructs of the generated code. The problems with this approach are similar to the problems with the time limit approach. Another option is terminating individuals based on heuristics such as memory read/write patterns.

6.2 Denial of service

Another issue with at least the same importance as termination is assuring that the genetic individual does harm the host system by consuming too many system resources, such as memory. If the genetic individual is to perform any non-trivial work, chances are it will need to allocate memory to work with. This includes direct allocations such as local variables and more indirect allocations like using data structures such as lists and maps. Once again, we can ask ourselves: What is to stop genetic code from consuming all available memory on the host system, crashing or freezing the host system? It would be trivial to evolve an individual doing nothing but looping around a memory allocating instruction.

Using Java, there is an easy solution. It is possible to limit the maximum memory usage of a Java application[4] using the JVM command line option "-Xmx". Should the genetic code cause more memory to be allocated than the limit allows, a JAVA.LANG.OUTOFMEMORYERROR (or similar) will be thrown, and the application will die.

This means that the genetic code needs to be run in another JVM than Genejector itself, or it would take down Genejector with it. I already implied in section 6.1 that the genetic individuals must run *in their own thread or process*. I narrow down this conclusion to *their own process*, as all threads of a JVM will die with it when any one thread dies from a JAVA.LANG.OUTOFMEMORYERROR.

Genejector thus launches the risen JVM for scoring. This is done by wrapping and reusing the original command line of the project to which Genejector is attached. After opening a communications channel between the risen JVM and the realm JVM, the original command line of the project is unwrapped and its main method is executed. When the project code at some point calls GENEJECTOR.GENEJECT(), a fresh individual will be requested from the realm JVM, executed and scored. The result will be sent back to the realm JVM through the communications channel. This process is repeated until the risen JVM eventually crashes (most likely because of a faulty individual) or is terminated by the realm JVM (most likely because a termination criterion has been reached or the individual got stuck as discussed in section 6.1).

With this elaborate isolation put into place, no matter how ill-behaved the individual is, the damage should be limited to the overhead cost of terminating and launching a new JVM.

6.3 Malicious actions

Yet another thing to assure is that genetic individuals do not harm the host system. One important example is access to the file system. Should the genetic code, directly or indirectly, find its way to the file system of the host system, there is nothing to stop it from deleting everything stored there. This would be a catastrophe for the average software developer.

Preventing this from happening using C/C++ or similar language executed directly on the host system is very difficult. Even if you do not include file system related genetic material in the gene pool, there are other ways a program could gain access to the file system. Examples worth mentioning are writing to arbitrary memory locations and buffer overflows. Both are very hard to prevent in C/C++ given the direct memory access that is so fundamental to the language(s). While it is highly unlikely that genetic code actually manages to pull off a malicious action, one has to keep in mind that it is theoretically possible and that many thousands (if not millions) of individuals may be generated during a run, each with a tiny chance of doing so.

Once again, the JVM becomes a great advantage. The genetic code will be sandboxed within the JVM (as any other Java application), and only be allowed access to the host system through the various Java APIs (such as JAVA.IO.FILE for file system access). By simply avoiding/blacklisting the unwanted/unsafe APIs from the

[4]not counting Java Native Interface (JNI) which allows Java code to call non-Java code outside the control of the JVM.

gene pool, there should be no way for the genetic code to harm the host system (unless it finds a security hole in the JVM[5]). On top of that, Java supports running applications under supervision by a *security manager*. A Java security manager enforces a *security policy* that states which system resources a Java application can have access to. It is, for example, possible to specify which files an application can read and write from/to and whether the application may create and use network sockets. Should an application violate the security policy, the offending action will be cancelled and a JAVA.LANG.SECURITYEXCEPTION will be thrown.

As previously mentioned, Genejector is designed to only use whitelisted APIs. It is thus up to the user to be the judge on whether APIs are safe for genetic programming or not.

[5]which would be worth enough money on the exploit black market to compensate for any damage caused.

7 Example output

The example output in figure 14 is obtained by testing Genejector against the included TripleCopyExample using the following command line[6]: *java -cp genejector.jar genejector.examples.TripleCopyExample*

```
Genjector: Request to bind to project received. Starting...
Dwayna: Starting run with the following settings:
 | Score limit:           6
 | Population size:       100
 | Generation limit:      100
 | Execution timeout:     10,000 (ms)
 | Status interval:       5,000 (ms)
 | Random seed:           -4598582820987392139
 | Project main class:    genejector.examples.TripleCopyExample
 | Project problem class: genejector.examples.TripleCopyProblem
 | Breeding operators:
 | | 16.00 -> 76%: MutationBreedingOperator
 | |  4.00 -> 19%: AddStatementBreedingOperator
 | |  1.00 ->  5%: RemoveStatementBreedingOperator
 | Approved breeding classes:
 | | Without instantiation:
 | | | genejector.examples.TripleCopyProblem
 | | | java.lang.Integer
Kormir: Created initial gene pool:
 | <Instruction>:
 | |  [G3] NewVariableGene [java.lang.Integer]
 | |  [G6] MethodGene      [genejector.examples.TripleCopyProblem#setOutput1(java.lang.Integer)]
 | |  [G9] MethodGene      [genejector.examples.TripleCopyProblem#setOutput2(java.lang.Integer)]
 | | [G12] MethodGene      [genejector.examples.TripleCopyProblem#setOutput3(java.lang.Integer)]
 | | [G13] MethodGene      [java.lang.Integer#toString()]
 | | [G14] MethodGene      [java.lang.Integer#hashCode()]
 | | [G15] MethodGene      [java.lang.Integer#compareTo(java.lang.Integer)]
 | | [G16] MethodGene      [java.lang.Integer#byteValue()]
 | | [G17] MethodGene      [java.lang.Integer#shortValue()]
 | | [G18] MethodGene      [java.lang.Integer#intValue()]
 | | [G19] MethodGene      [java.lang.Integer#longValue()]
 | | [G20] MethodGene      [java.lang.Integer#floatValue()]
 | | [G21] MethodGene      [java.lang.Integer#doubleValue()]
 | | [G22] MethodGene      [genejector.examples.TripleCopyProblem#setInputs(java.lang.Integer, java.lang.Integer, java.lang.Integer)]
 | <Statement>:
 | |  [G4] InstructionGene []
 | genejector.examples.TripleCopyProblem:
 | |  [G1] UseFieldGene    [problem]
 | java.lang.Integer:
 | |  [G2] MethodGene      [genejector.examples.TripleCopyProblem#getInput1()]
 | |  [G5] MethodGene      [genejector.examples.TripleCopyProblem#getOutput1()]
 | |  [G7] MethodGene      [genejector.examples.TripleCopyProblem#getInput2()]
 | |  [G8] MethodGene      [genejector.examples.TripleCopyProblem#getOutput2()]
 | | [G10] MethodGene      [genejector.examples.TripleCopyProblem#getInput3()]
 | | [G11] MethodGene      [genejector.examples.TripleCopyProblem#getOutput3()]
Kormir: Skipped all fields, methods and return values involving these non-approved classes:
 | boolean
 | byte
 | double
 | float
 | int
 | java.lang.Class
 | java.lang.Object
 | java.lang.String
 | long
 | short
Dwayna: Created initial population of mortal [M1] with root gene [G23] (StatementListGene).
Dwayna: Evolutionary breeding commenced!
Dwayna: [M2]: problem.getInput2().intValue(); [Score: 0]
Dwayna: [M3]: java.lang.Integer var1=problem.getInput3(); [Score: 0]
Dwayna: [M4]: problem.setOutput3(problem.getInput3()); [Score: 2]
Dwayna: [M5]: problem.getOutput3().longValue(); [Execution failure]
[...]
Dwayna: Solution [M1384] with a score of 6 found. Bred 1382 mortals in 00:00:10. Termination reason: Score limit reached.
Solution: problem.getInput3().hashCode();problem.setOutput1(problem.getInput1());problem.setOutput3(problem.getInput3());problem.setOutput2(problem.getInput2());
```

Figure 14: Output of an example run against TripleCopyExample (included in figure 4).

[6] Java sometimes cannot locate the *tools.jar* library from the JDK containing the Java compiler – even when the JDK is correctly installed. This issue can be worked around by copying tools.jar from the 'lib' folder of the JDK to the 'lib' folder of the JRE.

References

[1] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008.

[2] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993.

[3] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, 2010.

[4] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth A. De Jong, and Una-May O'Reilly. Genetic programming needs better benchmarks. In *GECCO*, pages 791–798, 2012.

[5] Simon M. Lucas. Exploiting reflection in object oriented genetic programming. In *EuroGP*, pages 369–378, 2004.

[6] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[7] The ECJ owner's manual. `http://www.cs.gmu.edu/~eclab/projects/ecj/docs/manual/manual.pdf`. Accessed: 28/02/2013.

[8] Genetic programming with JGAP. `http://jgap.sourceforge.net/doc/genetic_programming.html`. Accessed: 28/02/2013.

[9] GenPro. `https://code.google.com/p/genpro/`. Accessed: 28/02/2013.

[10] W. B. Langdon. *Data Structures and Genetic Programming*. PhD thesis, University College, London, 27 September 1996.

[11] The Java language specification, third edition. `http://docs.oracle.com/javase/specs/jls/se5.0/html/j3TOC.html`. Accessed: 03/03/2013.

[12] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.