Self-Programming Mutation and Crossover in Genetic Programming for Code Generation

Code-Generierung durch Selbst-Entwicklung von Mutation und Crossover in Genetischer Programmierung

Bachelor-Thesis von Viktor Nikolas Pfanschilling aus Gießen

Oktober 2017

Self-Programming Mutation and Crossover in Genetic Programming for Code Generation
Code-Generierung durch Selbst-Entwicklung von Mutation und Crossover in Genetischer Programmierung

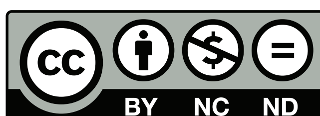
Vorgelegte Bachelor-Thesis von Viktor Nikolas Pfanschilling aus Gießen

1. Gutachten: Dr. Elmar. Rückert.
2. Gutachten: Prof. Jan. Peters. PhD.

Tag der Einreichung:

Please cite this document with:
URN: urn:nbn:de:tuda-tuprints-68996
URL: http://tuprints.ulb.tu-darmstadt.de/id/eprint/6899

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de



This puplication is licensed under the following Creative Commons License:
Attribution – NonCommercial – NoDerivatives 4.0 International
http://creativecommons.org/licenses/by-nc-nd/4.0/

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 22. Oktober 2017

(Viktor Nikolas Pfanschilling)

Thesis Statement

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, October 22, 2017

(Viktor Nikolas Pfanschilling)

Abstract

In Genetic programming, the genetic operators mutation and crossover are defined by the developer in order to produce a system that automatically generates computer code, usually incorporating information about the target language as well as about the problem. The genetic operators are a critical design decision because they control the evolution of genomes. This thesis outlines a way of eliminating that design decision by parameterizing genetic operators as part of the genome. In doing so, we discovered that several measures are necessary to accommodate the lack of assumptions that can be made about genomes. Among these measures are the introduction of an ancestry tree of genomes as additional bookkeeping to enable backtracking, the employment of a method for selection of candidates from that structure and the manual implementation of the first genome.

In our experiments, we discovered that -when uncompensated for- our approach will tend to bloat the genome extremely and distort our metrics. The cause of this was that individuals changed their own code to better fit the way their code generator works. We were able to solve this though, by employing only crossover, thus shifting the responsibility of improvement solely on the code generator's behavior. Our variant of crossover passes one genome's code to another genomes mutation function. We also discovered that the exact implementation of the initial genome is extremely important and seemingly inconsequential changes like removing a few lines of dead code make a huge difference. The results show that our current setup is still insufficient but demonstrates that our measures are effective.

Zusammenfassung

Genetische Programmierung wird zur automatischen Generierung von Programmcode verwendet. Dabei werden vom Entwickler die genetischen Operatoren mutation und crossover vorgegeben, die Genome modifizieren. Dabei wird Information über die Programmiersprache der Genome und oft auch das vorliegende Problem verwendet. Die Implementierung dieser Operatoren ist eine kritische Designentscheidung, weil sie beeinflusst wie sich die Genome entwickeln. Wir wollen in dieser Thesis eine Möglichkeit darlegen, diese Designentscheidung zu eliminieren, indem wir diese Operatoren als Teil des Genoms auffassen. Dabei haben wir festgestellt, dass mehrere Maßnahmen nötig sind, um dem Mangel an sicheren Annahmen über den Inhalt der Genome gerechtzuwerden. Teil dieser Maßnahmen sind die Einführung eines Stammbaums der Genome um backtracking zu ermöglichen, der Einsatz einer Auswahlmethode für Genome aus diesem Stammbaum und die händische Implementierung eines Ausgangsgenoms.

In unseren Experimenten haben wir entdeckt, dass die ungestörte Entwicklung der Genome selbige extrem aufbläht und unsere Messungen verzerrt. Das liegt daran, dass Individuen ihren Code so anpassen, dass er möglichst leicht durch ihren eigenen Codegenerator bearbeitet werden kann. Um dieses Problem zu lösen, haben wir nur crossover verwendet, sodass wir den Zwang zur Verbesserung ausschließlich dem Verhalten des Codegenerators auferlegt haben. Unser crossover gibt einem Genom den code eines anderen genoms. Wir haben auch festgestellt, dass die genaue Implementierung des ersten Genoms außerordentlich wichtig ist, da auch scheinbar vernachlässigbare Änderungen wie das Entfernen von totem Code große Änderungen haben können. Unsere Ergebnisse zeigen, dass unser derzeitiger Versuchsaufbau zwar unzureichend ist, aber die angewandten Maßnahmen zweckdienlich sind.

Contents

1. Introduction 2
1.1. Related Work . . . 2
1.2. Motivation . . . 3
1.3. Open Problems . . . 3
1.4. Outline . . . 4

2. Parameterized Mutation and Crossover in Genetic Programming 5
2.1. Fitness evaluation and type . . . 5
2.2. Backtracking and population structure . . . 5
2.3. Selection criterium . . . 6
2.4. Safety concerns . . . 6
2.5. Initial Genetic Operators . . . 7
2.6. Local Maxima . . . 7

3. Results 8
3.1. First Experiment . . . 8
3.2. Second Experiment . . . 11
3.3. Third Experiment . . . 13

4. Conclusion 16
4.1. Outlook . . . 16

Bibliography 19

A. Initial genome 21

B. Fitness type 22

C. Features for the Interestingness metric 23

D. Partition Problem Instance Generation 24

Symbols and Operators

List of Symbols

Notation	Description
$children$	the total number of direct children of a specific individual
$compilingChildren$	the number of successfully compiling direct children of a specific individual
$fitness$	the average fitness value of a specific individual, averaged over all problems
i	list of integers representing partitioning problem input
$interestingness$	a metric for our interest in evaluating a specific individual
o_x	xth list of integers representing partitioning problem result

List of Operators

Notation	Description	Operator
$\mathrm{fit}_{\mathrm{PP}}$	the fitness value of a partitioning problem solution given as input list and a tuple of output lists	$\mathrm{fit}_{\mathrm{PP}}(\bullet, \bullet)$
Σ	the sum of all elements of a list	$\Sigma_\bullet$

1 Introduction

Genetic programming is a tool used in the automated generation of computer code. In it, the genetic operators mutation and crossover are commonly defined by the developer. These two operators modify genomes of source code and incorporate prior knowledge about the genome language. Additional assumptions about the problem can be incorporated. The implementation of genetic operators is thus a critical design decision because they dictate the way genomes evolve. This thesis outlines a way of eliminating that design decision by parameterizing genetic operators as part of the genome.

1.1 Related Work

Genetic Programming, in general, is a means of automatically generating programs that maximize a given fitness function. These programs are randomly generated; the ones that achieve the highest fitness are mutated randomly and merged with other genomes. Usually, this happens in a strictly iterative fashion: The set of active genomes, called population, gets expanded by new genomes and reduced again through selection. Thus, the state of the system "consists only of the current population of individuals in the population" [1, p. 14]. In general, both the *mutation* operator as well as the *crossover* operator are implemented manually and are static. Static operators are difficult to define. They depend on many constraints such as the target language, the problem and the expected performance, and are usually purpose-built for the target language [2].

There are different, related approaches in the field of evolutionary algorithms[3]. In general, all of these subforms share some common ground drawn from biology: solutions are treated like individuals in a population, their offspring will generally differ from them in a minor way and only the fittest - according to some criteria - are allowed to reproduce[4]. The result is an algorithm that gradually optimizes a solution of some type according to the implementation of the fitness criterion and the implementation of the operators. Due to the nature of this overall design, the usually most costly step - evaluation of new individuals[5] - can be parallelized[6][4].

Haskell is a purely functional programming language with strict, static, implicit typing. This is useful for genomes since the compiler provides a very specific filter of invalid genomes. There exists work on genetic programming in Haskell, using "type information to reduce the search space" [7]. Its type inference features appear to be useful for this[8]. Another approach[9] used a custom-built strongly typed programming language to narrow the search space.

According to our knowledge, there is no previous research on using a parameterized code generator in genetic programming. Several of our core components and issues though have already been researched. Firstly, [10] researched the implications of the proportion of mutation and crossover. They found that the difference usually isn't that great, contrary what we will observe in our setup. Secondly, code growth in genetic programs (bloat) is well researched, at least purely as a side effect of accumulated noise. There exist approaches to measure [11] and avoid[12][13] bloat. They do not seem to apply directly to the type of bloat we will be seeing in this thesis. Furthermore, [14] researched the effects of various selection schemes in the context of a noisy fitness function.

1.2 Motivation

Genetic programming, compared to many other machine learning methods like neural networks, has the benefit of delivering a somewhat readable output, their performance is however somewhat lacking for most practical applications. The goal of this thesis is to explore the feasibility of implementing the code generator, i.e. the mutation operator, within the genome itself. This variant of genetic programming would have one fewer design decision than regular genetic programming. It also would achieve improved speed of optimization via adapting the code generator to the chosen language and given problem.

Ideally, a genetic program with such a structure could learn faster the more it is trained. This could make genetic programming more competitive in modern machine learning. Thus a high standard of success of our approach would be achieved if our algorithm could optimize problems faster the more we let it solve.

1.3 Open Problems

Encoding genetic operators within the genome introduces several new challenges to genetic programming. The three most salient are:

(i) First, it implies abandoning all assumptions about the genome and the contents of the genome can no longer be reasoned about. This brings safety and performance concerns with it. Unsafe code could have unintended consequences such as crashing or performing undesired actions. Incorrect code could not compile in the first place, which provides no guarantee that a set of new individuals contains any useful information.

(ii) Individuals can not be evaluated directly and definitely. The most important measure, quality of the code generator - whether that is defined by average or peak quality of children genomes, or compilation rate of child genomes - cannot be directly measured; the code can only be evaluated by means of an attempt at compilation. Even then the resulting fitness value is tainted by noise. Fundamentally, our evaluation can only be so accurate: On average, we can only have one compiling offspring per genome, because that compiling offspring is another individual to be assessed. In consequence, criteria are needed by which to allocate testing time to individuals of interest.

(iii) It is not guaranteed that a properly developed line of genomes which developed well will continue to do so in future iterations. For instance, a genome might jam its own code generator, at which point this particular line of genomes has reached its peak performance. Genomes might also update source code with trivial changes only, resulting in a flood of working, albeit useless genomes.

1.4 Outline

In chapter 2, we will go over the measures taken to alleviate the anticipated issues and problems discussed in section 1.3. These measures include our two most important contributions, an ancestry tree of genomes as well as a selection criterion to pick candidates from this tree. Afterwards, we will present the experiments we performed in chapter 3 and discuss the results. We performed three experiments that illustrate the importance of the choice of selection criterion and initial genome. We will then go over the implications of our results and provide an overview of possible improvements in 4.

2 Parameterized Mutation and Crossover in Genetic Programming

In this chapter we will present the measures we took to solve the problems associated with learned mutation and crossover operators.

2.1 Fitness evaluation and type

In order to deal with the possibility of non-functional genomes, several steps were necessary. First, we decided to implement a more expressive fitness type that encodes non-compilation or compilation, as well as a fitness value. With this type, non-compilation can be expressed as failing at one of several stages - like syntax checking or type checking. Additionally, compiling individuals are tagged with a fitness value achieved on computational problems. Currently, this is the NP-complete Partitioning Problem.The Partitioning Problem is, in our case, defined as:

$$fit_{PP}(i,(o_1,o_2)) = \begin{cases} 0, & \text{if no permutation} \\ 1 - \frac{|\Sigma_{o_1} - \Sigma_{o_2}|}{\Sigma_i}, & \text{if permutation} \end{cases}$$

Note that the fitness values are normalized to be between 0 and 1.

The final fitness value is the average of the fitness over all problems considered.

2.2 Backtracking and population structure

Conventionally, genetic programming happens in terms of strict generations, where every generation is discarded upon generating the next one. This is impractical, because there are no guarantees about the performance of the new generation, as the code-generation behavior needs to be tested first. It is entirely possible that the entire new generation has a defective code generator and our entire population would then die out.

To mitigate this, individuals are stored in a tree structure to enable backtracking. Here, individuals are stored as a dependant of the code-generating parent. This structure contains an ID, fitness and an optional file path to the source code of the individual. The source code is subject to garbage collection. Using this approach, it is possible to keep a hundred thousand genomes in store. Storage of the tree without the source codes is cheap at more than 10000 individuals per megabyte. This structure also helps if a generation did not manage to achieve an improvement over the previous.

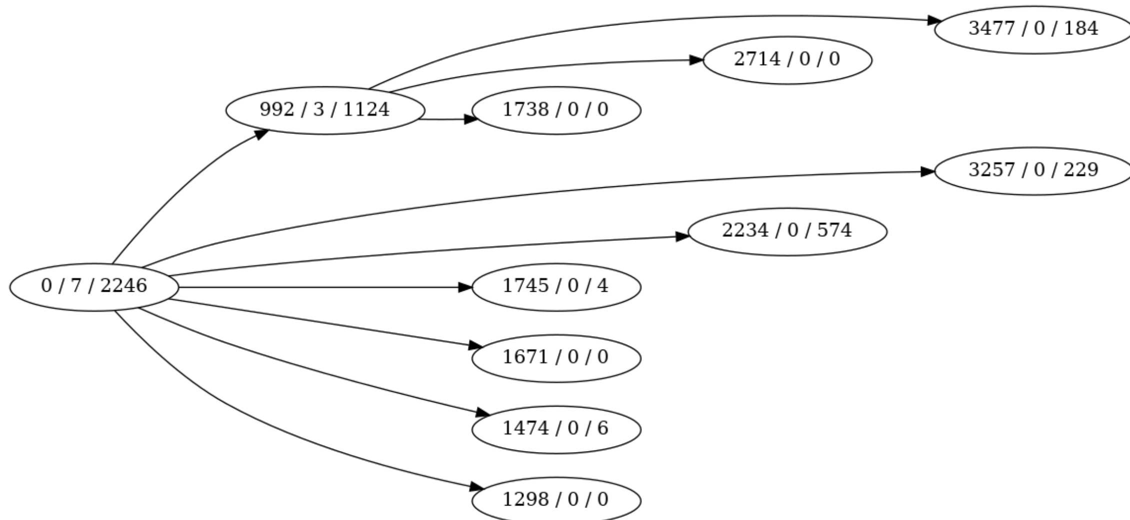


Figure 2.1.: Example Population Structure. Generations are vertically aligned. Labels represent the chronologic IDs, the number of successfully compiling offspring and the total number of offspring. Non-compiling individuals are not pictured. Note that, in our approach, backtracking allows us to draw our current generation from offspring from all previous generations, whereas in standard Genetic Programming, the latest generation is only drawn from the last.

2.3 Selection criterium

The concept of storing all past generations opens up new possibilites. Instead of being tied to the last generation, we can now decide which genomes to pick for reproduction from the entirety of previous individuals. To determine which genomes in the tree are interesting enough to be evaluated further, we use an easy-to-exchange interestingness metric that determines both when backtracking should take place as well as which genomes to replicate. This metric is defined as a function of, among other features, generation, fitness, compilation rate of children and the number of children. From this, the interestingness metric computes a real-valued interestingness, which is used when assigning tickets for the next generation. Tickets are assigned proportionally to computed interestingness, i.e. for every iteration there exists a constant and having an interestingness of n-times that constant equates to receiving n tickets. That constant is then determined so that the desired amount of tickets will be assigned.

2.4 Safety concerns

Since the program will be executing randomly generated code, we implemented a safety constraint: The type of the functions to be learned is defined ahead of time to exclude input or output (I/O) action. This way, code in one genome can not influence other genomes or other processes on the system. Additionally, Safe Haskell[15] could be used. In this subset of Haskell, the compiler guarantees that no undeclared I/O occurs. This would prevent modules from being imported if they contain unsafe I/O, i.e. I/O effects in functions with pure, I/O-free signatures.

2.5 Initial Genetic Operators

Since the search space is extremely large and sparse, the initial genome was implemented by hand. It contains the initial implementation for crossover and mutation, both of which get existing source code in String encoding and are to return source code as a String. The initial implementation of mutation we used splits this String into keywords and arbitrarily changes, deletes or inserts keywords from a dictionary. It then uses a library-provided Haskell syntax checker to validate the result and start over if the syntax check fails.

To address crossover of genomes A and B, we pass A's code into B's mutation function. This is a relatively simple option in that it does not require the merging of actual source code, but it can not be considered optimal because of exactly that reason. While the exact assignment of source codes to mutation functions is determined on a per-experiment basis, the actual implementation of crossover is genome-specific; in fact, it is the exact same code as the mutation operator.

2.6 Local Maxima

In order to avoid local maxima, we mark as local maximum every genome that has a suspiciously high compilation rate of its offspring. To do this, we observe the compilation rate gain over the genome's parent and the genome's number of children. The rule applies to all genomes with high number of children, thus high confidence, and a high increase in compilation rate. These genomes and any descendants will no longer be considered. Typically, these kinds of genomes tend to crowd out the remaining population while contributing nothing to the problem. This rule is not ideal, as we do not know the classification error because there is no ground truth test for this property.

3 Results

In this section, we will show that the measures described in the previous section are effective. We will conduct three experiments with different configurations to illustrate the most important design decisions. All three experiments have in common that they were conducted as a sequence of 1000 generations, 100 individuals each. We used the Partitioning Problem as fitness function, and used

$$interestingness = \frac{compilingChildren + 1}{children + 1} + \frac{2}{100} * fitness$$

as interestingness metric. This interestingness metric ensures that confidently low compilation rates are dropped while all others are kept, with a bias towards unconfident, high compilation rates.

3.1 First Experiment

For our first trials, a seed genome was used that modifies its own code and tries to ensure that its syntax is valid, i.e. we are working with *mutate* only. This approach achieved a total compilation rate of 9%, where approximately 50% of the non-compiling genomes are due to type errors, with most of the rest being caught in various stages before type checking.

Figure 3.1 shows the compilation rate of the children of individuals.

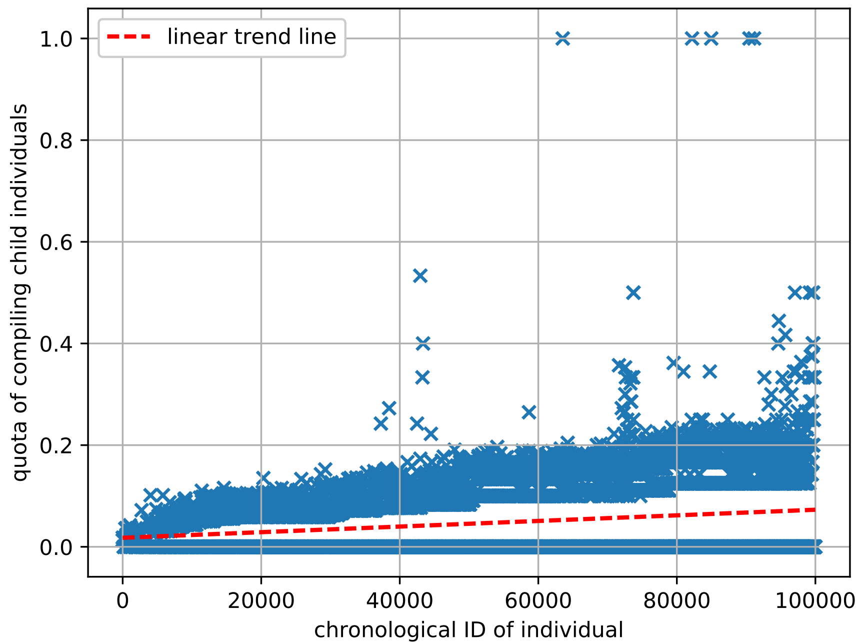


Figure 3.1.: Quota of compiling children by chronological genome ID. Note that outliers are included in this figure. Note that given a compilation rate of 9%, the compilation rate of any individual can only be sampled about 11 times on average, thus exact compilation rates are subject to sampling errors.

The compilation rate of the most recent genomes grows as a function of time, due to overfitting to the problem: Manual inspection shows that those genomes have merely added dead code. For example, genome 91627 contained the segment 3.3 where genome 0 contained the segment 3.2 originally. Note that the original segment was dead as well, and the replacement, albeit completely senseless, is actually valid code, and it is relatively easy to find a valid modification.

```
typechecks str =
        case parseModule str of
                ParseOk ast -> typecheck ast ;
                otherwise -> False
```

Figure 3.2.: Initial state of the dead code segment

```
p3 typechecks act foldl div p2 p3 str p4 func p1 foldr not filter =
        case str p1 div p2 not p2 act filter div foldl foldr act p4 p2 p3 func
        foldl func not div act div act map foldl p2 foldr not map not not not
        filter div not p2 foldr filter func foldr act filter p1 map act foldr
        p2 p3 div foldl foldl p3 p1 p2 func map p4 not func div act act $ func
        foldr p3 foldl foldl filter p4 p3 not div p4 foldl filter not not
        p3 not not of
                ParseOk ast -> p3 foldr ast . foldl p2 foldr ;
                otherwise -> p1 func p2 p3 p3 foldr . act False not p3 filter . foldl
```

Figure 3.3.: Exemplary state of the dead code segment at the end of the experiment.

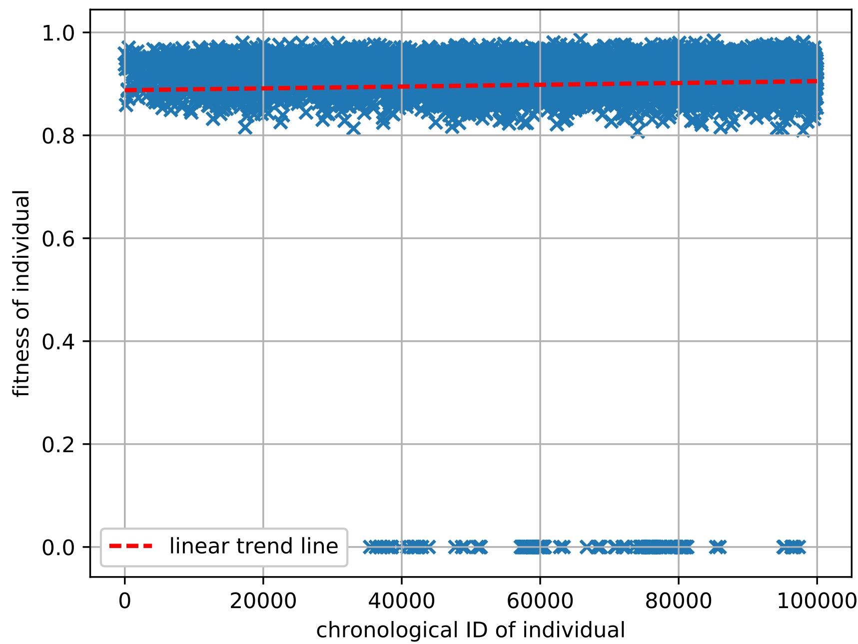


Figure 3.4.: Fitness function by chronological genome ID. The outliers with a fitness value of zero are usually caused by runtime errors during fitness evaluation.

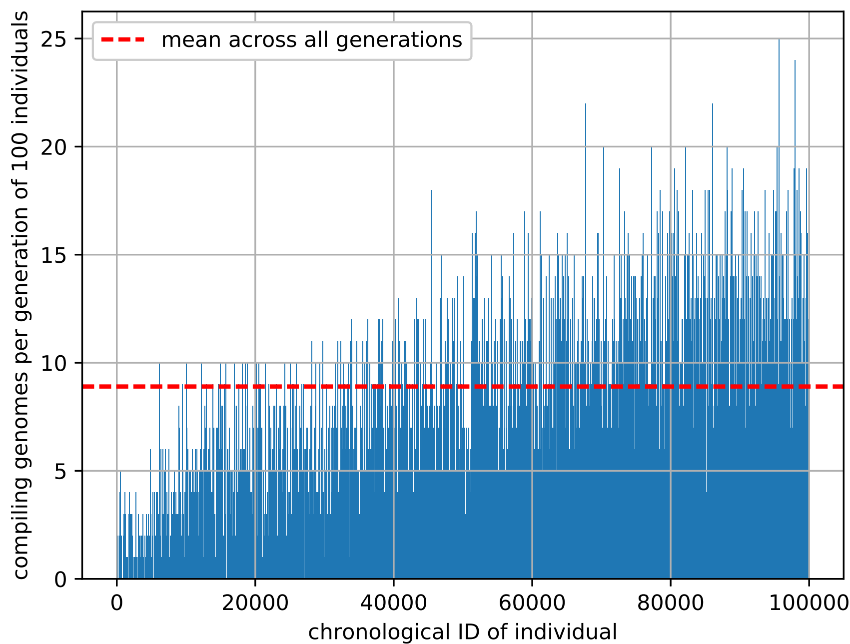


Figure 3.5.: Number of compiling genomes per generation of 100 genomes. The rising trend is due to unintended overfitting of the code's structure to the code generator, i.e. the way the code is written is being changed to achieve better compilation of mutations, while the mutation algorithm remains unchanged. This happens mostly by adding dead code.

Removing the dead code segment from the genome results in a compilation rate of 3%, drastically reducing the speed of overfitting. In this instance, 1% of the initial genome's offspring compiled and manual inspection shows that the increase in compilation rate is again attributable to non-semantic changes in source code, not changes in behavior.

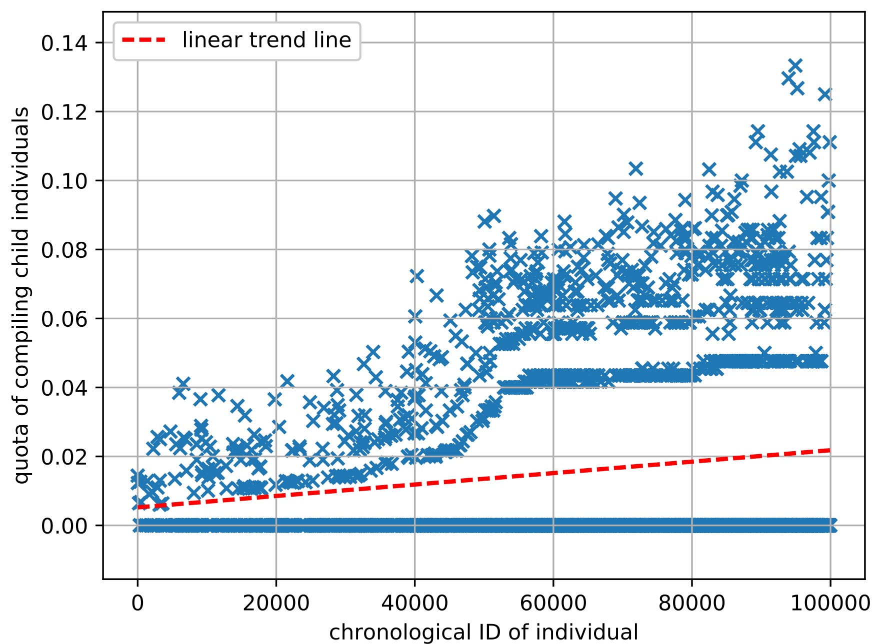


Figure 3.6.: Quota of compiling children by chronological genome ID. There appear to be no outliers in the data. The maximum compilation rate was much lower than in experiment 1, thanks to less overfitting. The remaining increase is still -at least largely- attributable to overfitting.

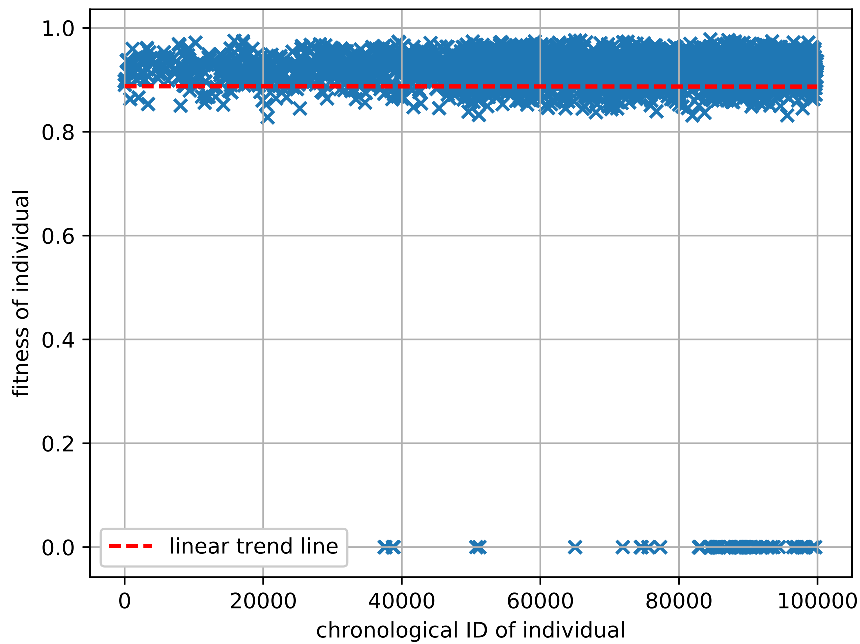


Figure 3.7.: fitness function by chronological genome ID. The proportion of 0-fitness outliers has increased from 1 in 45 in the previous experiment to 1 in 30. This probably shares a common cause with the lower number of compiling genomes. It is likely caused by an overall higher chance of harmful mutations.

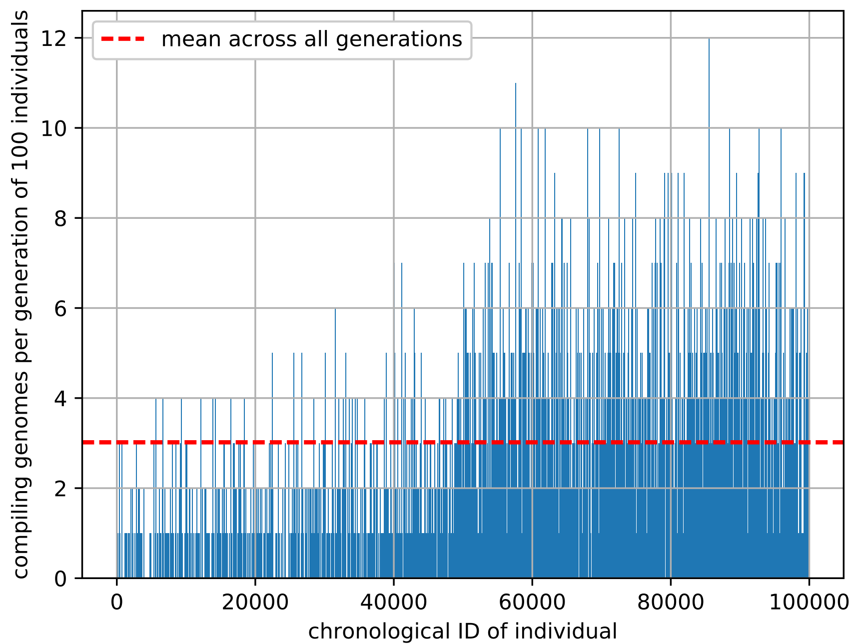


Figure 3.8.: Number of compiling genomes per generation of 100 genomes. The rising trend is due to unintended fitting of the code's structure to the code generator. The mean is significantly lower than in experiment 2.

In a next experiment, we attempted to alleviate this issue by using our previously described *crossover* only, i.e. every genome is generated by applying the code generator of one genome to the code of another. The compilation rate is only tracked for the code generator-supplying parent. This way, making one's own code easily modifiable is not a valid strategy and changes to the code generator's behavior are necessary.

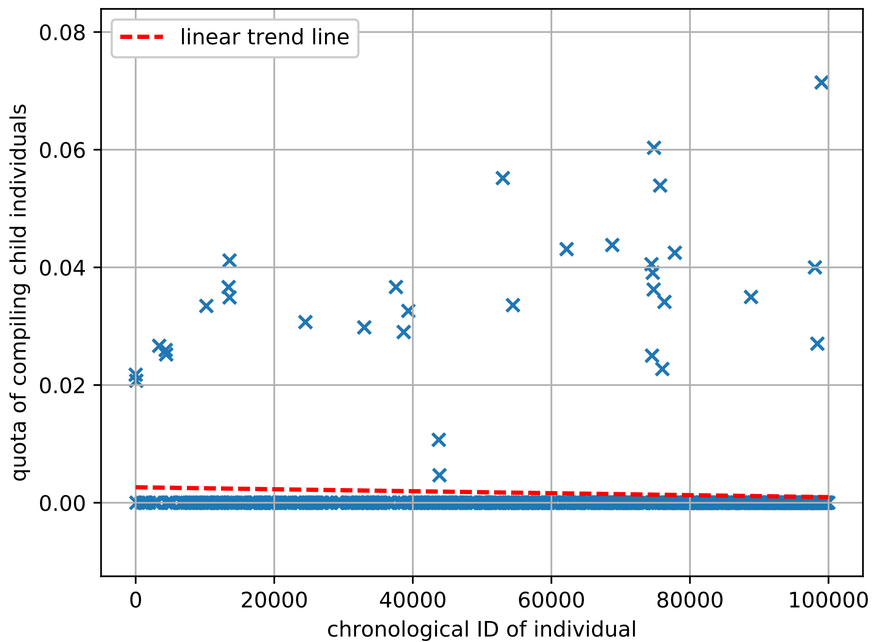


Figure 3.9.: Quota of compiling children by chronological genome ID. Note that outliers are included in this figure. Also note the slightly declining trend line. This might be because the search heuristic is getting more "desperate", including lower quality genomes more as its exploration of the better branches concludes, which could indicate that too little exploitation is occurring.

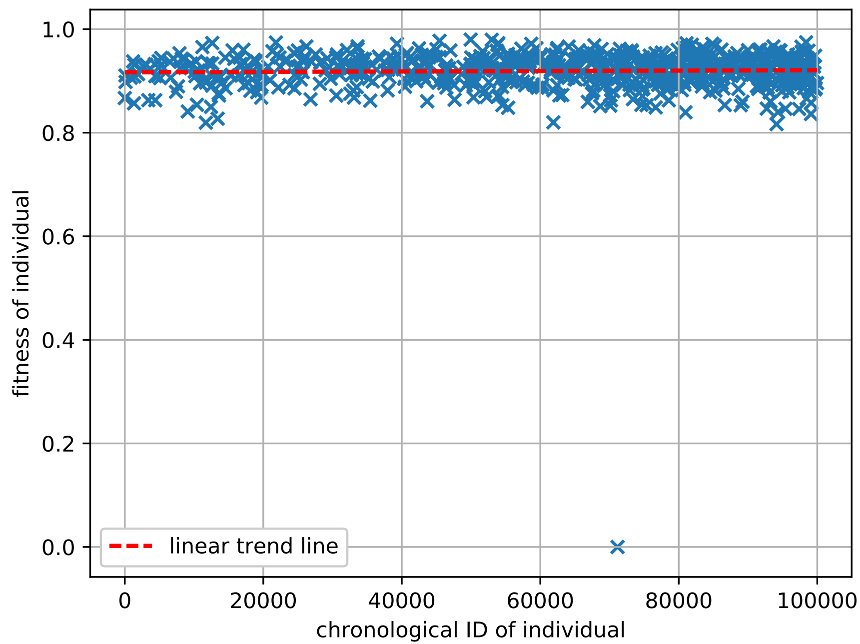


Figure 3.10.: fitness function by chronological genome ID. Curiously there are almost no genomes with broken partitioning-problem code, while plenty exist that have a broken code generator. This could be because the used initial implementation of the partitioning problem solver is hard to damage without violating its type. This can not be said about many auxiliary functions of the code generator.

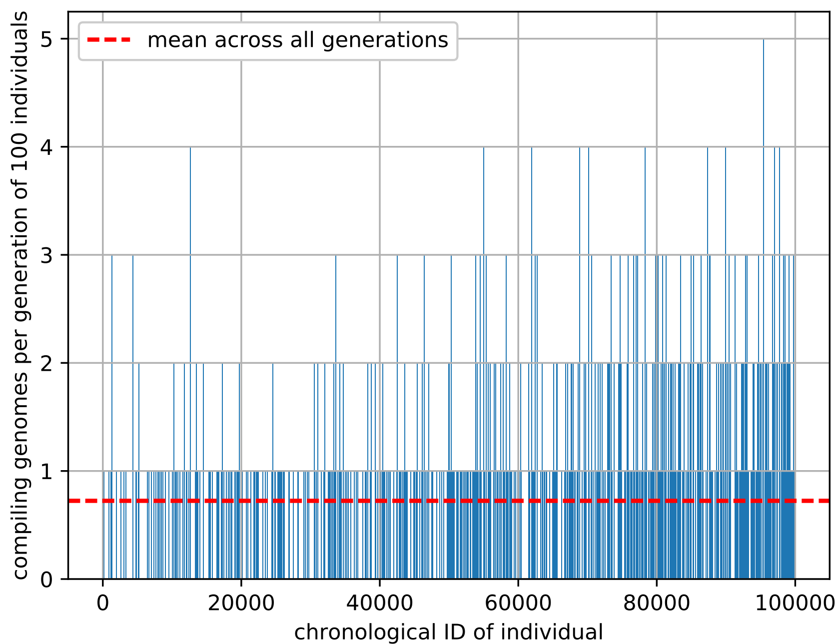


Figure 3.11.: Number of compiling genomes per generation of 100 genomes. Note the drastically lower mean, which indicates that our efforts to control the amount of overfitting are successful.

The results were mixed. Generally, results vary from trial to trial, as the amount of actually functional semantic changes is generally extremely low and thus its effects are noisy. In the above trial, compilation rates decreased strongly compared to previous experiments because the code could no longer adapt to the code generator. Over the course of the trial, compilation rates stayed nearly constant, increasing only slightly. Overall, the compilation rate was at 0.72%.

Interestingly, most of the failed genomes actually are failures of their parents to execute properly, i.e. most failures occur as runtime errors in the code generator. These broken code generators are relatively clearly separated from working genomes: There are 595 genomes that never succeeded, failing 56843 times at generating any code. Contrary to that, the 52 genomes that never had runtime failures generated code (compiling or not) 25436 times. Note that taking longer than 10 seconds to generate code was presumed to be an infinite loop and thus counted as a runtime error. Contrary to an initial observation of the data, these numbers suggest that the undesired property of runtime errors is in fact being filtered out by the interestingness metric, it is just really common.

4 Conclusion

Manual inspection of source code showed that individuals in our first two experiments developed dead code. This appears to be because any change to dead code has a lower likelihood of causing damage than a change to live code, thus dead code increases compilation rate. Since we explicitly selected high compilation rates through our interestingness metric, genomes with dead code were preferred. Even though there exist ways to detect [11] and avoid [12] bloat in genetic programming, since these approaches rely on the relationship between code growth and fitness growth to tell beneficial from detrimental bloat, they will not be able to solve this issue; after all, the bloat we saw contributed to fitness.

The results show that the goal was not reached, as undesired mutations drowned out desired ones. This was caused by an insufficient fitness function that did not attempt to capture the code generator's qualities and an interestingness metric that encouraged undesired traits. We can infer from the drastic difference of removing the dead code in 3.2 that results are extremely dependent on the exact code of the initial genome and that the right choice of initial genome is a critical decision in making our approach work.

Our results show that, as of now, we did not overcome the problems that accompany a mutable genetic operator. It can, however, be shown that some problems were solved. The tree which is used to store individuals does its task of handling non-compiling genomes and backtracking well. Enough information is retained to build a interestingness-metric upon. Our crossover implementation was able to eliminate the relationship between an individual's code structure and the compilation rate of its offspring. It seems that we have pushed back the overfitting to a tolerable level.

4.1 Outlook

Possibly the most crucial aspect that could be improved is the seed genome. The most promising approach would be to leverage the *Language.Haskell.Exts.Syntax*[16] module to manipulate code directly within the syntax of haskell. This would drastically lower the complexity of implementing improvements like typechecking in the genome. Fortunately, this module has an interface that is completely free of I/O, thus merging well with our safety design. This is not the case across the board, as many libraries that deal with Haskell code are heavily contaminated with I/O-effects.

A current open problem is crossover. It is hard to merge two programs in the general case. We decided to implement crossover of genomes A and B by handing to A's mutation function the code of B. This does not merge genomes on the source level. An alternative would be passing A's code in as well, which would offset the benefits this method had on code structure dependent compilation rate. Merging the two ideas, we could try passing to A the code of two other genomes

B and C. Provisions for passing in multiple sources exist in the type of the mutation function.

It should also be noted that our fitness function is very noisy, due to the way we implemented the problems. Currently, every individual gets a number of individually generated problem instances of given sizes, which are all given the same weight. This was done to prevent overfitting to the specific problem instances but has the unintended consequence that more instance-dependent problem sizes are weighted the same as more stable longer sizes, causing a lot of noise. A clearer optimization target could be achieved by providing a weaker initial solution and smoother problem instance weighting. Alternatively, one could use the same instances for all individuals, but swap those every few generations and recompute all fitness values.

On a more technical level, several aspects of our implementation could be refined, possibly with drastic performance benefits. One possibility, although limited, is the application of automatic testing approaches to genome evaluation. There is a popular library [17] that handles automated, randomized property testing and is being used successfully[18]. However, it generally only handles pass-fail distinctions and would have to be adapted to assign fitness values. More importantly though, we're dealing with source code a lot, and currently we evaluate and generate our genomes by compiling and calling genomes from the command line. This leaves much to be desired, as compiled genome executables can be big and execution times are slow. This issue could be alleviated by making use of Haskell's extensive set of libraries, among which is plugins[19], a library that handles compilation, interpretation and runtime loading of haskell source code. This could speed up our experiments as a whole, but by how much is hard to say without testing it. Most of our program is already parallelized, but some parts remain that operate on the whole ancestry tree and have not been parallelized. Potentially, these algorithms could be optimized as well.

The detection of undesired genomes could be expanded. Currently, individuals that learn to make trivial changes to foreign source code are encouraged. This could be alleviated by using an interestingness measure that imposes some liveness criterion on offspring, for example requiring that fitness or compilation rate values increase with a given certainty. This is, however, difficult to implement, as we have a regression problem on a tree-structure. Currently, we only extract the features for the interestingness metric within a limited horizon. From a slightly different perspective, one could view the interestingness problem as one of exploration versus exploitation. Also, for the purposes of the interestingness metric, we do not differentiate between different kinds of non-compiling. If we were to retain more information about compilation, we could reinforce type errors more than syntax errors, as a type error implies lack of syntax errors. This could for example be done by computing the risk posed by every particular compiler stage.

Bibliography

[1] J. R. Koza, *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science Stanford, CA, 1990.

[2] P. Nordin, "A compiling genetic programming system that directly manipulates the machine code," *Advances in genetic programming*, vol. 1, pp. 311–331, 1994.

[3] D. B. Fogel, "An introduction to simulated evolutionary optimization," *IEEE transactions on neural networks*, vol. 5, no. 1, pp. 3–14, 1994.

[4] M. Tomassini, "Parallel and distributed evolutionary algorithms: A review," 1999.

[5] S. Harding and W. Banzhaf, "Fast genetic programming on gpus," in *European Conference on Genetic Programming*, pp. 90–101, Springer, 2007.

[6] N. Hansen, S. D. Müller, and P. Koumoutsakos, "Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es)," *Evolutionary computation*, vol. 11, no. 1, pp. 1–18, 2003.

[7] T. Yu and C. Clack, "Polygp: A polymorphic genetic programming system in haskell," *Genetic Programming*, vol. 98, 1998.

[8] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *Journal of functional programming*, vol. 17, no. 1, pp. 1–82, 2007.

[9] D. J. Montana, "Strongly typed genetic programming," *Evolutionary computation*, vol. 3, no. 2, pp. 199–230, 1995.

[10] S. Luke and L. Spector, "A revised comparison of crossover and mutation in genetic programming," *Genetic Programming*, vol. 98, no. 208-213, p. 55, 1998.

[11] L. Vanneschi, M. Castelli, and S. Silva, "Measuring bloat, overfitting and functional complexity in genetic programming," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pp. 877–884, ACM, 2010.

[12] T. Soule, J. A. Foster, and J. Dickinson, "Code growth in genetic programming," in *Proceedings of the 1st annual conference on genetic programming*, pp. 215–223, MIT Press, 1996.

[13] R. Poli, "A simple but theoretically-motivated method to control bloat in genetic programming," *Genetic programming*, pp. 43–76, 2003.

[14] B. L. Miller and D. E. Goldberg, "Genetic algorithms, selection schemes, and the varying effects of noise," *Evolutionary Computation*, vol. 4, no. 2, pp. 113–131, 1996.

[15] "Safe Haskell in the glasgow haskell compiler users guide." `https://downloads.haskell.org/~ghc/7.8.4/docs/html/users_guide/safe-haskell.html`. Accessed: 2017-10-19.

[16] "haskell-src-exts library." `https://hackage.haskell.org/package/haskell-src-exts-1.18.2`. Accessed: 2017-10-18.

[17] "QuickCheck testing library manual." `http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html`. Accessed: 2017-10-18.

[18] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," *SIGPLAN Not.*, vol. 46, pp. 53–64, May 2011.

[19] "Plugins haskell library." `https://hackage.haskell.org/package/plugins-1.5.6.0`. Accessed: 2017-10-18.

A Initial genome

The entirety of the code can be found on **https://github.com/vektordev/GP**

This genome was written to be more easily modifiable by itself. This harms the readability somewhat.

```
reprogram :: [StdGen] -> State -> [String] -> (String, State)
act :: [StdGen] -> State -> Input -> (Output, State)
--Active/mutable part of genome starts here.
reprogram ( r1 : _ ) state ( source1 : _ ) = let candidates = map ( \ rng -> lexemlisttransform (
    preproc source1 ) rng state ) ( infrg r1 ) in (head $ filter ( \ candidate -> ( candidate /=
    postproc ( preproc source1 ) ) && ( parseable candidate ) ) (map postproc $ filter (\x -> True)
     candidates), state )
parseable str = let result = parseModule str in wasSuccess result
wasSuccess ( ParseFailed _ _ ) = False
wasSuccess ( ParseOk _ ) = True
preproc str = concat $ intersperse ["\n"] $ map words $ lines str
postproc strs = rmlist ( \ x y -> x == '\n' && y == ' ' ) $ unwords strs
infrg rg = let ( x , y ) = split rg in x : infrg y
rmlist predicate ( x : y : ys ) = if predicate x y then rmlist predicate ( x : ys ) else x : rmlist
     predicate ( y : ys )
rmlist a xs = xs
initial = [ 10000000 , 20000000 ]
lexemlisttransform [] rng state = []
lexemlisttransform ( lex : lst ) rng state = let ( decision , rng2 ) = next rng :: ( Int , StdGen )
     in if decision < ( head initial ) then let ( n , rng3 ) = next rng2 in ( lexems !! ( mod n $
    length lexems ) ) : lex : ( lexemlisttransform lst rng3 state) else if decision < ( last
    initial ) then lexemlisttransform lst rng2 state else lex : ( lexemlisttransform lst rng2 state
    )
act rngs state ( PPI inp ) = ( PPO ( take ( div ( length inp ) 2 ) inp, drop ( div ( length inp ) 2
     ) inp ) , state)
```

B Fitness type

Note that *fitness* refers to the Float contained in the type Fitness and assumes a fitness-level of at least compilation.

```
data Level =
  Unchecked |
  Unsafe |
  RuntimeErrOnParent |
  MiscErr |
  ParseErr |
  ScopeErr |
  AmbiguousSymbolErr |
  TypeErr |
  NoBindingError |
  UnknownCompilerError |
  Compilation deriving (Show, Read, Eq, Ord)

type Fitness = (Level, Float)
```

C Features for the Interestingness metric

This structure again assumes compilation.

```
data FeatureVec = FeatureVec {
  id :: Int
, parentid :: Int
, state :: State --is the genome active, inactive or discarded
, isLocalMax :: LocalMaxState --is the genome considered a local maximum
, generation :: Int -- distance from initial genome
, fitness :: Float
, fitnessGainSinceParent :: Float
, compilationRate :: Ratio Integer
, compilationRateGain :: Ratio Integer
, children :: Int
, avgChildFit :: Float
} deriving (Show, Read)
```

D Partition Problem Instance Generation

```
computeProblemFitness :: ([StdGen] -> State -> Input -> (Output, State)) -> State -> IO (Float,
    State)
computeProblemFitness actFnc agState = do
  let queries = [2..11]
  rngs <- replicateM (length queries) newStdGen
  inputs <- mapM generateInput [2^n | n <- queries]
  let (newSt, outs)= foldr (\ (input, rng) (state, oldouts) -> let (actOut, actState) = actFnc [rng]
       state input in (actState, actOut : oldouts)) (agState, []) (zip inputs rngs) :: (State, [
     Output]) --actFnc [rng] agState input
  fits <- zipWithM fitness inputs outs
  return (mean $ zipWith normalizeFitness inputs fits, newSt)

generateInput :: Int -> IO Input
generateInput 0 = return $ PPI []
generateInput n = do
  x <- getStdRandom next
  (PPI xs) <- generateInput (n-1)
  return $ PPI (x:xs)
```