Amaru - A Framework for Combining Genetic Improvement with Pattern Mining

Oliver Krauss
oliver.krauss@fh-hagenberg.at
University of Applied Sciences Upper Austria
Hagenberg, Upper Austria, Austria

ABSTRACT

We present Amaru, a framework for Genetic Improvement utilizing Abstract Syntax Trees directly at the interpreter and compiler level. Amaru also enables the mining of frequent, discriminative patterns from Genetic Improvement populations. These patterns in turn can be used to improve the crossover and mutation operators to increase population diversity, reduce the number of individuals failing at run-time and increasing the amount of successful individuals in the population.

CCS CONCEPTS

• **Software and its engineering** → **Search-based software engineering**; *Interpreters*; • **Theory of computation** → **Genetic programming**.

KEYWORDS

Genetic Improvement, Compiler, Interpreter, Framework

ACM Reference Format:
Oliver Krauss. 2022. Amaru - A Framework for Combining Genetic Improvement with Pattern Mining. In *Genetic and Evolutionary Computation Conference Companion (GECCO '22 Companion), July 9–13, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3520304.3534016

1 INTRODUCTION

Amaru[12] (http://amaru.dev) is a framework that enables research in Genetic Improvement (GI), and learning from GI experiments by mining frequent patterns that occur in the population of the evolutionary experiments. These patterns can be applied to discover novel optimization techniques, or to improve the genetic operators in future GI experiments. In doing this, Amaru follows two core goals, *enabling GI at a level close to the compiler* and *identifying and explaining recurring patterns* in the source code.

Other frameworks in this domain exist, most notably GinTool [5, 22] and PyGGI [1, 2]. Compared to Amaru, both of these frameworks are lightweight, and easy to use. Amaru is less lightweight, as it directly integrates with the Truffle[23] interpreter and the Graal[19] compiler. This enables Amaru to directly access compilation information not available in other frameworks, such as the stack and heap information, as well as a rather fine granular view of the source code, as the representation utilized in GI is the Abstract Syntax Tree (AST) directly interpreted by Truffle and compiled by Graal. This also comes at a disadvantage compared to the other frameworks. While the fine granular representation allows for smaller changes to the source code, the search space is much larger, as even simple test-languages have hundreds of different node types, with complete languages such as JavaScript having more than 2,000, many of which need specific considerations in the evolutionary operators, making Amaru more heavyweight. Amaru is currently only extensible to programming languages implemented in the Truffle framework, and these will execute on the Java Virtual Machine (JVM).

In addition to enabling GI research, Amaru stores the information generated during GI experiments in a knowledge base. This enables mining these results, as well as producing reproducible experiments that may be shared with fellow researchers. The focus of this mining is on identifying frequent, discriminative patterns, i.e. frequent patterns that occur more often in one part of a population than in another. As an example, this allows identifying interesting patterns, such as sub-ASTs that frequently occur in individuals of the GI population that have a higher run-time performance compared to the original source code, and does not occur in those ASTs with a lower run-time performance, thus implying that the pattern is responsible for the improvement. To validate such identified patterns, Amaru provides a validation mechanism to prove or disprove identified patterns. This shows a first step in the direction of explainable genetic improvement, which is an important topic in the GI community [15].

The remainder of this publication is structured as follows. section 2 gives a short overview of the underlying technologies of Amaru, which is explained further in section 3. Details on how GI is applied in Amaru is discussed in subsection 3.1, and pattern mining and verification is explained in subsection 3.2. Finally an outlook is provided in section 4.

2 BACKGROUND

In the following we shortly outline the Graal Compiler, as well as the Truffle Interpreter, which are used as the basis for Amaru.

2.1 Graal

Graal is an aggressively optimizing just-in-time (JIT) compiler, written in Java as part of the OpenJDK project [19]. It compiles Truffle ASTs to efficient machine code. It features multiple optimizations, such as inlining, loop unrolling and partial escape analysis. Some of these optimizations are speculative, i.e. based on heuristics. These

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '22 Companion, July 9–13, 2022, Boston, MA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9268-6/22/07…$15.00
https://doi.org/10.1145/3520304.3534016

optimizations can be taken back, in a process called deoptimization, and other optimizations are applied. This often happens if the execution context changes, for example when different branches become the hot paths. Graal uses an Intermediate Representation (IR), which is a directed graph consisting of the control flow and the data flow, and builds the basis for research in compiler optimizations. Amaru does not modify the Graal IR, but rather only modifies the ASTs that Truffle utilizes. The use of Graal necessitates special consideration towards how non-functional features, such as run-time performance, are measured for individuals in GI populations, as its aggressively optimizing nature requires a high warm up time of 100,000 iterations [8, 16, 20].

2.2 Truffle

Truffle is an open-source framework and self-optimizing interpreter for the prototyping of programming languages, called guest languages. It uses Abstract Syntax Trees (AST)s as a representation form. Truffle itself is written in Java and can run on any JVM, but directly integrates with Graal, enabling high performance compilation of Truffle guest languages. Truffle itself contains additional optimizations such as node specialization and loop optimization. Languages implemented in Truffle have all advantages of the JVM, such as garbage collection. Currently, there are several open-source language implementations of Truffle, including JavaScript, C, Python and Ruby. Truffle guest languages can also interface with each other. For example, a JavaScript Truffle node can produce a call to a C Truffle node [9, 23, 24].

Every node in a truffle AST represents a concept of the language, such as "write int to stack", "double / double". In some cases, a single node is not enough, and Truffle injects optimization nodes. For example, the while loop consist of three nodes, "while", "loop" and "repeating". In this case, the "repeating" and "while" nodes are provided by the language developers, while the "loop" node is provided by the Truffle framework, enabling truffle to analyze and optimize the AST. A simplified example of a truffle AST representing a recursive Fibonacci sequence is shown in Figure 1. It shows the granularity that is enabled by directly manipulating such ASTs with GI.

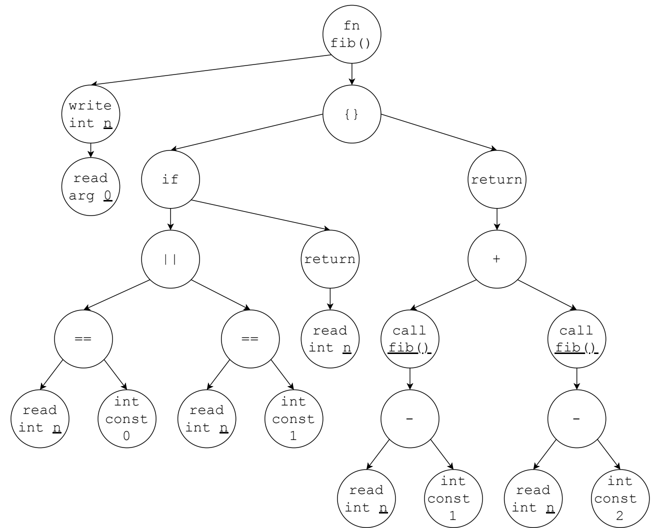

Figure 1: AST representation of a recursive implementation of the Fibonacci sequence.

3 ARCHITECTURE OVERVIEW

The architecture of Amaru is shown in Figure 2. It is intended to be run directly in the Graal Virtual Machine (VM), which is a part of the OpenJDK, and can interface with *Guest Languages* implemented in Truffle. A guest language is essentially a language that was prototyped via the Truffle Framework. In the framework, every operator or operand, e.g. loops, switches, variables, literals, etc. is implemented as a node class. By using features from the Truffle API, this enables Truffle to optimize generated ASTs, and Graal to compile them natively in the JVM.

The framework consists of two essential parts. The first part (top right of Figure 2) deals with GI experiments in a Truffle guest language. The second part of the framework (bottom left in Figure 2) enables mining patterns from GI experiments and enables analyzing these patterns further with additional experiments.

Both of these are connected via a *Knowledge Base*, e.g. a Neo4J graph database, which stores the *Experiment Results* generated during the GI experiments. This includes every AST that was created as well as observed run-time information, such as occurred exceptions during different tests, or the run-time performance. Which genetic operator created the ASTs, and via which parent ASTs, is also stored. For example, in a crossover operation, both of the parent ASTs are connected with the child AST. This enables tracing genealogies during the experiments and analysis of the impact that different operators have during the experiments. *Truffle Language Information* on the Truffle guest language is also stored in the knowledge base, such as which operators and operands are available, and assertions about them, e.g. if they conduct a reading or writing access to stack or heap, or if they are branching statements etc. This information can in turn be used both by GI operators or considered when mining patterns.

The optimization side of the framework allows creating *Experiments*. These experiments consist of a program that is run in the Truffle guest language, and one function in the program that is under optimization. Optimizing multiple functions at the same time is currently not supported. The validation of these functions can be done via unit tests, only testing the function directly, or integration tests in which the entire program, or a benchmark function is run. The tests and benchmark corpus currently need to be provided by a researcher, although automated test generation is a future research topic. Additional configuration of the experiments contains, for example, the fitness function, selected genetic operators and operands, or which patterns should be excluded or enforced during a GI run.

The *Optimizer* is the part of the framework that conducts the experiments, i.e. runs them. It supports using different algorithms and operators to guide the search during the experiment, with the primary algorithm being Knowledge-guided Genetic Improvement (KGGI). It also allows the connection to other frameworks, to either

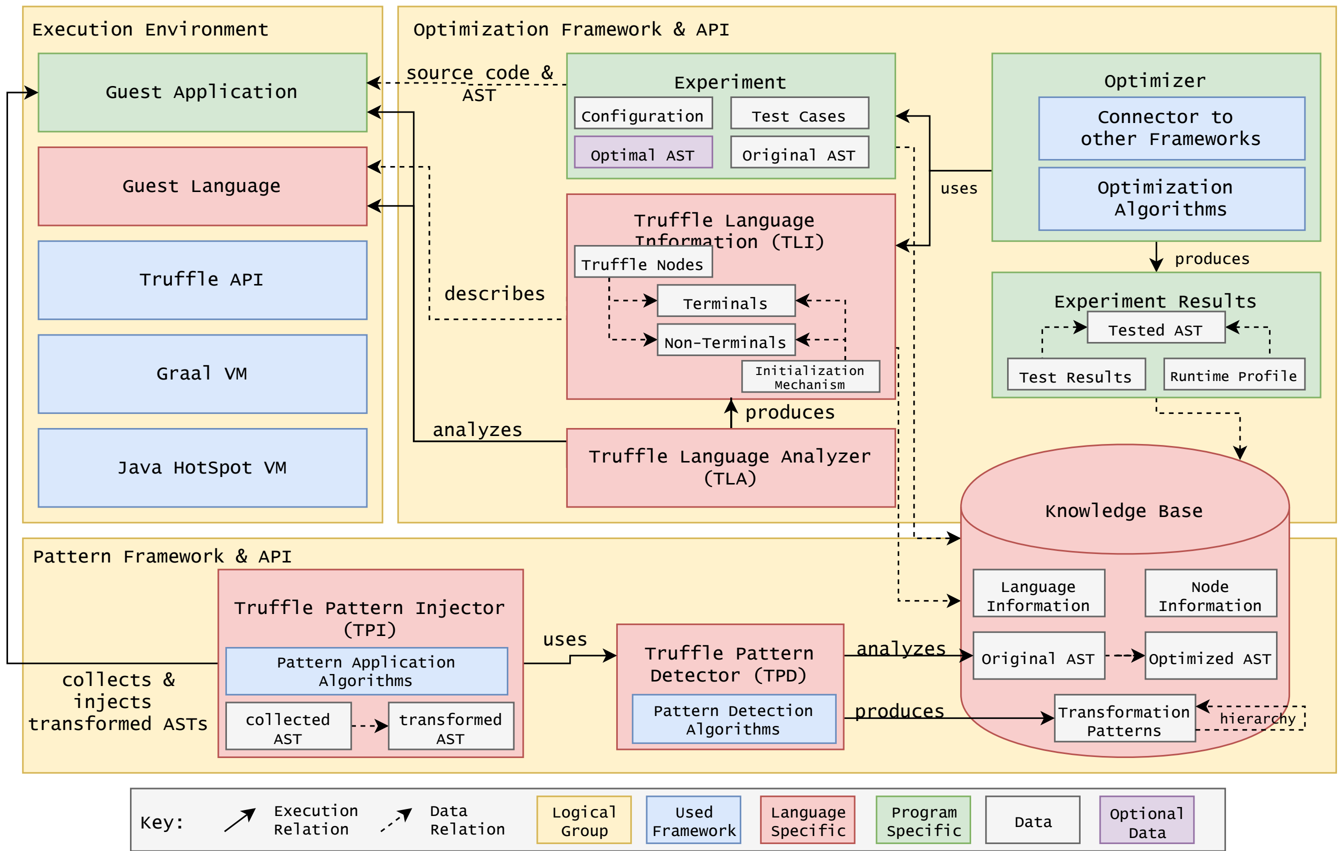

Figure 2: Architecture of the Amaru framework. It builds upon the Truffle and Graal execution environment, and consists of functionality for optimization using GI. The experiment data is stored in a knowledge base, from which patterns can be mined and verified [13].

outsource parts of the algorithm, such as a specific mutation or crossover implementation, or to enable complete control of the search from another framework, with Amaru only serving to compose the ASTs and running them on the Graal compiler. The only existing connector to another framework is to HeuristicLab [7, 11]. With this connection mechanism, Amaru can be a valuable addition for other frameworks and can serve as an intermediary, as it also contains mechanisms to automatically analyze and prepare Truffle Languages for use in machine learning.

The *Truffle Language Analyzer* serves to analyze Truffle languages and provide the *Truffle Language Information* via various mechanisms. Static code analysis, via Java reflection, is used to extract Truffle API specific information from nodes. Dynamic analysis is primarily conducted via brute force operations, that, for example, identify which variable write operations need to be in which order and which data types are interoperable. For example, in an untyped language writing a string to a variable and then reading it as an integer may be allowed, but in a statically typed language this will fail at runtime. Similarly, one language may require arrays to be allocated with a size, while another will dynamically allocate the array with the write operation, not requiring an allocation beforehand. The resulting information provides a generic representation, independent of the actual language, that enables the instantiation of AST nodes, and having information available about which nodes influence the control or data flow, and call other functions. Without this automated analysis, utilizing Truffle languages would be an overwhelming overhead for developers of a Truffle language, or those wanting to utilize it for GI, especially frameworks not written directly in Java.

Amaru features several options of running the different ASTs during the GI experiments, as shown in Figure 3. The simplest implementation is the internal executor, which simply injects the AST provided by a genetic operator in the context of the program and runs the given test suite. Other operators simply exist to load additional information from the ASTs. For example, the trace executor, automatically injects tracing information into the Truffle languages via byte code manipulation. This can be used to identify exactly which nodes of an AST are executed for a specific test case, or to measure the run time of single nodes or branches, thus identifying the hot paths for a single test, or test suite.

Due to the fine granularity, of how ASTs are designed in Amaru, ASTs that crash the JVM during their execution are a rare, yet regular, occurrence. Exceptions from which the JVM cannot recover,

for example, are when the garbage collector overhead limit is exceeded, or stack overflow exceptions. An even larger issue is the valid measurement of non-functional properties, especially runtime performance. It is widely known that the garbage collector and other processes running on the operating system can influence measuring runtime [4]. In a similar manner, the same code may behave differently on other hardware architectures, operating systems or just compiler version or flags used during the compilation.

A unique, and lesser known challenge when using an aggressively optimizing compiler, like Graal, and possibly affecting multiple modern compilers is code caching [20, 21]. In essence, code caching identifies code snippets that are executed often and caches them, including their optimizations. Modern compilers also have a warm-up time (for Graal the first 100,000 iterations) in which code is analyzed and (re-)optimized multiple times. However, in GI it is not unusual to conduct only 10 repetitions [6] when measuring runtime performance, due to the high cost of compiling and executing large code bases. In addition, individuals in a GI population are often very similar, primarily as GI deals with improving existing code instead of synthesizing new one, but partly because of the popular code grafting operators utilize [3].

Individually these three considerations, code caching, warm up, and a low amount of repetitions, may be less problematic. When considering them in combination, these individual ingredients become a recipe for disaster when attempting run-time performance measurements. Since GI features highly similar code, it is likely that the compiler caches it. Since the compiler also does not finish its warm up with the few iterations the code is run, it may appear that the GI algorithm is producing faster and faster individuals over several generations, while what is actually happening, is that the compiler continuously improves recurring snippets.

To enable accurate measurement of non-functional features of software, Amaru primarily applies remote execution, meaning that the program steering the search only generates the ASTs but never executes them. In Figure 3 the AST is run by remote runners (right). These connect with the Optimizer via a Broker (center), using the Message Queue protocol. The Broker takes the initial configuration of an experiment from the optimizer, i.e. the source code, and which features should be measured, and then distributes generated ASTs to the remote runners. Each of the remote runners is in their own JVM. Whenever runtime performance is measured, the runner shuts itself down after running just a single AST, with one single test case, to ensure that there is no bleed-over-effect between AST measurements. A control plane checks in regular intervals if the runner process still exists, and starts a new process if it does not. The control plane also monitors crashes, and can report the reason a runner stopped to the Broker. This makes Amaru resistant to hard crashes, and allows identifying which AST leads to such a type of crash. Via this mechanism, Amaru also allows for distributed execution, as multiple control planes can connect from different PCs to one broker. In the case of run-time performance, the broker instructs each control plane to only have one runner active, otherwise the control plane automatically spawns one runner per CPU thread.

The pattern mining side of the framework (Figure 2 bottom) can load data from one or multiple *Experiments*, and utilize the information obtained from the Truffle language. The *Truffle Pattern Detector* contains some algorithms to mine patterns. The primary implementation being a novel mining algorithm, Independent Growth of Ordered Relationships (IGOR), and a reporting tool to view the resulting patterns found in the ASTs. The *Truffle Pattern Injector* can then later inject patterns into the GI operators for future experiments, to either enforce that anti-patterns will not be mutated or crossed into the population anymore, or enforces the inclusion of patterns that should occur in the population. This concept is comparable to grafting [3], with the primary difference being that instead of grafting specific source code, more generic structures are injected, and often mutated to complete them, or crossed with various sub ASTs.

3.1 Genetic Improvement in Amaru

Genetic Improvement in Amaru is done at the AST level of Truffle. The Truffle guest languages are analyzed by Amaru, and processed into generic constructors, with meta information on how the nodes can be combined into an AST, e.g. which child nodes are valid. In addition, how nodes need to interact is also documented. For example, the Amaru framework detects which nodes initialize a variable correctly to be read by another node. In addition, the function registry, stack and heap variables and their assigned types are available for the genetic operators. This allows Amaru to guarantee in most cases that only such ASTs can be constructed, that they will always be able to compile. Some edge cases still exist, for example if the Truffle guest language is designed in a way where child node types cannot be inferred via static analysis.

Amaru uses Knowledge-guided Genetic Improvement (KGGI) as base algorithm [14]. KGGI is a combination of Grammar-guided Genetic Programming (GGGP) [18] since it adheres to the valid structure of the language, and Tree Genetic Programming (TGP) [10], as it uses the Truffle AST as representation.

At the core of KGGI stand two concepts. Knowledge about the language being optimized, and the syntax graph, an example of which is shown in Figure 4. The syntax graph is a graph consisting of strategies that can be applied in all major genetic operators, crossover, mutation and selection. The core concept of this graph is based on *requirements engineering*. Each (sub)-AST is seen as a set of requirements.

Consider the AST from Figure 1 being applied in the mutation operation. If the mutation operation were to decide that the *write int n* node needs to be removed, the syntax graph traverses the rest of the AST, and collects unsatisfied requirements. In this case, each of the *read int n* nodes will register that no corresponding node exists to initialize the stack variable, thus leading to an unsatisfied requirement. This unsatisfied requirement is then used when generating a new sub-AST, to enforce that the variable n must be initialized with any node able to write an integer value. This does not guarantee that no run time exception may occur. If, for example, the replacement sub-AST would initialize n with any negative integer, this would lead to a stack overflow exception, crashing the JVM.

The second concept of KGGI is the knowledge about the language, as is shown in the white boxes in Figure 4. Due to the language analysis by Amaru, it is known for each node, what their minimal requirements are concerning depth and width, e.g. how

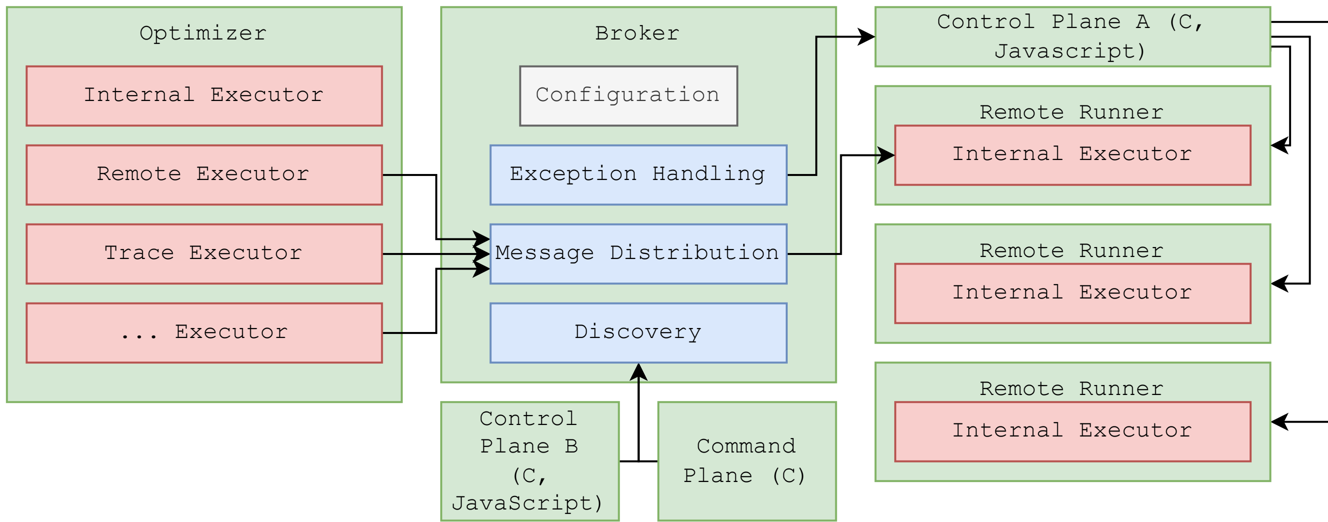

Figure 3: Available execution mechanisms for AST individuals in the population, enabling failure free, distributed execution and accurate measurement.

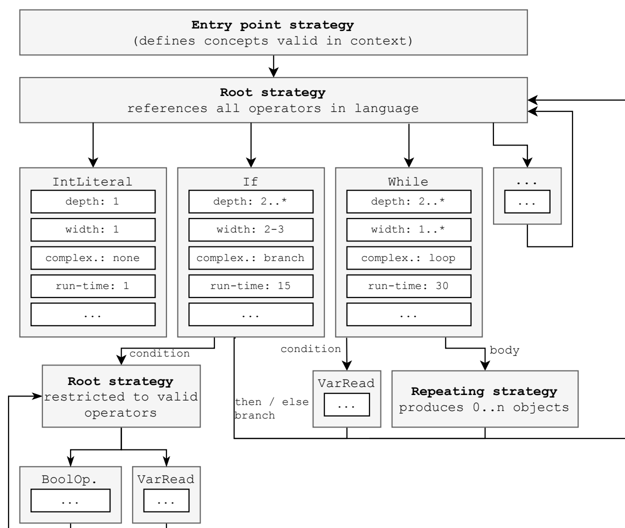

Figure 4: Syntax graph for a given Truffle guest language. Operators (gray) represent the grammar and contain knowledge about their non-functional properties (white), and edges to valid relationships according to the grammar. [13]

many child nodes must be created as a minimum valid AST. This lets the syntax graph control the search space and prevent code bloat. This also works with assertions about non-functional features, as is shown with run-time. The Truffle Language Analyzer, allows measuring the average cost of a node, to approximate the total cost of a generated AST. For branches and loops, assertions are made which branch is likely to be taken, or how often a loop will be executed. While this is by no means an accurate measurement, as it is highly dependent if these assertions hold true, and which optimizations the compiler will apply during run-time, it still serves to restrict the search space in the GI populations.

The entry point strategy serves the purpose of guaranteeing that the Grammar is upheld in the operations. For example, during the mutation operation, if this strategy is given the condition of an if statement, it will only generate a new sub-AST that returns a boolean value.

The root strategy serves purely as a selection mechanism for all other strategies. It recursively queries all strategies with the given requirements, for example that a variable must be written to, or that a maximum depth must not be exceeded. Strategies either fulfill requirements, or add new requirements, which in turn must be fulfilled by other child or sibling nodes.

All other strategies are more specialized, such as specific strategies for the stack or heap access, or strategies that deal with repetitions, such as multiple statements that may be created within a block statement. Most of these strategies are automatically generated from the information inferred from the Truffle guest language. Some others are provided by Amaru and automatically injected when flags exist, such as specialized strategies for stack or heap access, or strategies dealing with invocations of functions. Amaru allows the injection of manually created strategies as well, to handle edge cases, or for example support grafting via such a strategy, to still ensure that the graft only occurs at places allowed by the languages' grammar.

The syntax graph has the distinct advantage that it allows a fine granular control of the search space, and can guarantee that generated ASTs will contain nodes that are indispensable for the AST's functionality. This allows to significantly reduce the rate of run-time exceptions that will occur during execution. This can be even further reduced, since anti-patterns and patterns are also injected as requirements. For example, an anti-pattern may be that the Fibonacci call provides the wrong argument datatype, which

would lead to a run-time exception (see Figure 5). Such an anti-pattern can be introduced as a requirement, where if the function being called is "fib" then there must be an int argument provided. This in turn may still produce a run-time exception as the int variable must also not be negative, which is easily modeled for literals, but hardly possible to guarantee for every possible sub-AST that may produce an integer as the argument node. This rather simple example shows that patterns do help to improve overall quality of the GI populations, but cannot guarantee a runtime exception free population.

Amaru provides much tooling for the syntax graph, and enables the automated injection of patterns and anti-patterns which can simply be modeled as ASTs. Wildcards, allow these patterns to be fairly expressive, to also model the requirement that a node must (not) be there, or that nodes can be skipped. However, all of this also comes with a disadvantage. The evaluation of the syntax graph is a fairly expensive operation, since queries must be recursively iterated until either an AST is found, or the search space is exhausted as the (potential) AST becomes too large. While Amaru mitigates this issue with caching, and different selection strategies, the KGGI crossover and mutation operations are much slower than other operators known from literature.

3.2 Mining Patterns from Experiments

The Amaru framework provides functionality to mine ASTs produced from GI experiments via frequent and discriminative pattern mining. Mining frequent patterns enables the identification of substructures that occur frequently overall. E.g. patterns are not identified in one single AST, but rather patterns are identified occurring in a large percentage of ASTs of the entire GI population, making them *significant*. Discriminative pattern mining on the other hand compares two or more groups and attempts to find patterns that occur more in one group than in the other(s) making them *discriminative*. For example, frequent pattern mining is the process of discovering patterns over an entire GI experiment. Discriminative pattern mining occurs if the ASTs produced in the experiment are grouped into those succeeding all tests, those failing tests, and those producing run-time exceptions, which would allow a mining of bug patterns.

An example of a mined anti-pattern that is responsible for a bug is shown in Figure 5. The figure shows an invocation of the Fibonacci function with a char literal on the left, which will fail as no such function exists to be invoked. On the right is the corresponding correct pattern which will not fail as an int literal is provided. The invoke node and function literal node on both sides would not be discriminative, as they occur in both search spaces of failing and succeeding ASTs. However, the third node makes the pattern discriminative when they only occur in one group, or at least far more often.

Pattern mining deals with a search space 2^n, where n is the amount of relationships available in an AST. This is because each component of a tree, e.g. any subset of nodes with corresponding relationships between them, must be analyzed. As this search space is virtually unmanageable for larger ASTs multiple growth metrics exist [17], and a multitude of algorithms to apply these metrics exists as well. Metrics are applied to filter and rank patterns, and

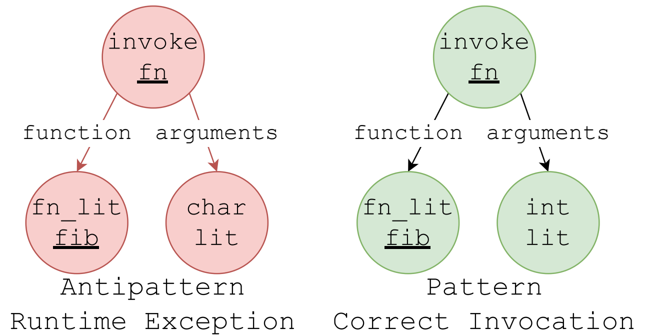

Figure 5: Example of an identified anti-pattern leading to a run-time exception as the Fibonacci function "fib" is invoked with the wrong type, and the corresponding identified pattern where the invocation correctly provides an integer.

only the top n patterns are analyzed for additional relationships. Amaru provides several reference implementations of such metrics, although primarily the *contrast* metric is used for discriminative mining. This metric prioritizes the difference in *frequence* between discriminative groups, e.g. patterns that occur more often in one than in the other, by absolute difference.

Amaru uses the Independent Growth of Ordered Relationships (IGOR) algorithm [13], which was developed to mine ordered relationships in ASTs. In mining ordered means that the relationships order is considered, as some algorithms will consider the pattern $(A) \leftarrow (B) \rightarrow (C)$ identical to $(C) \leftarrow (B) \rightarrow (A)$ as B is the parent node, and both A and C occur as child nodes. Another concept to be considered is induced vs. embedded. Induced patterns are such patterns where all nodes must be directly connected. Embedded patterns can skip nodes, only enforcing a direct connection, such as A and C would have.

IGOR provides only ordered mining, as the execution order of statements, and often even partial statements, such as boolean expressions which are short-circuit evaluated, is relevant in source code. It allows both induced mining, to identify larger connected patterns, as well as embedded mining to enable identifying patterns that may be distributed over the entire AST. This is done via a pattern growth approach. E.g. all patterns of size 1 are evaluated, and ranked according to the selected metric. From this point on only the top n patterns are checked, and only grown if an additional relationship and corresponding node would remain discriminative enough with appropriate support, e.g. the pattern also still occurring often enough in at least one group.

The IGOR algorithm also allows utilizing the Truffle guest languages' information in the mining process, by introducing the concept of hierarchies, e.g. generalizing observed nodes. In the example shown in Figure 5 the correct pattern could be an argument of any "int" instead of just int literals. As another relevant example, mining algorithms other than IGOR would not be able to consider for and while loops as the same pattern. IGOR can apply the natural hierarchy of truffle languages to automatically generalize patterns. Alternatively a hierarchy can be provided instead that targets specific use cases, such as hierarchies prioritizing the data

types of nodes, or hierarchies prioritizing the data flow, for example generalizing "read int", and "read double" to a "read" and further generalizing "read" and "write" to "data access". This provides two advantages. Firstly, generalized patterns have more options for their application. Secondly, it allows mining general patterns combined from a multitude of specialized patterns that may not occur often enough to be discriminative on their own, but are significant and discriminative when combined.

Amaru provides functionality to verify or disprove identified patterns. This is done via using the mutation operator of KGGI and enforcing the pattern in multiple mutants of one or more ASTs. The effects of the AST are then measured, e.g. tests are run and metrics are observed. From this, a confidence score is calculated that relays in percent if the expected hypothesis holds true, or not.

As an example, the anti-pattern from Figure 5 is assumed to be responsible for leading to a run-time exception. If this pattern is injected 100 times into different ASTs, the confidence score is calculated by how often the AST fails during run time (ex. 95%). Due to the complexity of code, patterns can rarely be proven to be 100% correct, as the pattern may be injected in a branch that is not executed with the given test suite, or it may be injected at unreachable positions, for example after a return statement. However, this method enables the verification of patterns and fixes for these patterns. Patterns that have been verified, can then be applied in future GI runs to exclude patterns leading to runtime exceptions, or alternatively to introduce well performing genomes.

4 CONCLUSION AND FUTURE WORK

Genetic Improvement is a research field that is still growing. One of the current challenges in this domain is building trust in GI, and making the generated results, explainable [15]. Amaru provides a first step in this direction by utilizing the data generated during GI runs, and applying discriminative pattern mining on the results.

Amaru runs via Truffle on the JVM, enabling the use of any language developed as a Truffle guest language. This enables research on GI on a fine granular level, and utilizing information from interpreter and compiler, such as stack and heap information. Compared to other GI frameworks, such as Gin and PyGGI, Amaru is less lightweight due to its integration with Truffle.

Amaru is quite successful in combining GI with pattern mining, and we hope that this success can be replicated by the community. Krauss[13] provide a case study consisting of 25 algorithms from three different domains (math, sort and neural networks). They apply pattern mining to these experiments, and identify several bugs that cause most of the GI populations to lead to run-time exceptions. They prove that these bugs are responsible for their corresponding exceptions with an average confidence of 90.1%, and when applying these highly confident patterns in a further experiment series. In their results they manage to double the population diversity, as the large number of failed ASTs reduced that diversity significantly, and reduce the number of individuals failing at runtime to 36.9 % over all experiments. Additionally, they find improvements for 22 out of the 25 selected algorithms, with an average run-time speedup of 33.5%.

The source code of Amaru is publicly available at https://github.com/oliver-krauss/amaru. The connector to Heuristic Lab is available at https://github.com/oliver-krauss/heuristiclabconnector. Both are licensed under the Mozilla Public License 2.0. It also enables the integration with other frameworks, and extensibility for additional algorithms, and operators for both GI and pattern mining. Participation is welcomed, and encouraged, be it in the form of feature-request and bug reports or contribution to the code base.

In the near future, we plan to improve and extend Amaru. This primarily concerns rewrites to ease the set-up and allow users to quickly start their own experiments. Both the GI and pattern mining algorithms currently allow reporting in the form of generated static HTML or Markdown reports after the experiments have finished. No UI is available for configuring experiments or viewing interim results as of yet. We also plan to extend support for more algorithms from search based software engineering.

REFERENCES

[1] Gabin An, Aymeric Blot, Justyna Petke, and Shin Yoo. 2019. PyGGI 2.0: Language Independent Genetic Improvement Framework. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Tallinn, Estonia, 1100–1104. https://doi.org/10.1145/3338906.3341184

[2] Gabin An, Jinhan Kim, Seongmin Lee, and Shin Yoo. 2017. PyGGI: Python General framework for Genetic Improvement. In *Proceedings of Korea Software Congress (KSC 2017)*. Busan, South Korea, 536–538. https://coinse.kaist.ac.kr/publications/pdfs/An2017aa.pdf.

[3] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. [n. d.]. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014-11-11) *(FSE 2014)*. ACM, 306–317. https://doi.org/10.1145/2635868.2635898

[4] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. [n. d.]. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of the ACM on Programming Languages* 1 ([n. d.]), 52:1–52:27. Issue OOPSLA. https://doi.org/10.1145/3133876

[5] Alexander E. I. Brownlee, Justyna Petke, Brad Alexander, Earl T. Barr, Markus Wagner, and David R. White. 2019. Gin: Genetic Improvement Research Made Easy. In *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, Prague, Czech Republic, 985–993. https://doi.org/10.1145/3321707.3321841

[6] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. [n. d.]. Injecting Shortcuts for Faster Running Java Code. In *2020 IEEE Congress on Evolutionary Computation (CEC)* (Glasgow, United Kingdom, 2020-07). IEEE, 1–8. https://doi.org/10.1109/CEC48606.2020.9185708

[7] Daniel Dorfmeister and Oliver Krauss. [n. d.]. Integrating HeuristicLab with Compilers and Interpreters for Non-Functional Code Optimization. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (Cancun, Mexico, 2020-07). ACM. https://doi.org/10.1145/3377929.3398103

[8] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *2nd Asia-Pacific Programming Languages and Compilers Workshop (APPLC'13), as Part of the 10th Annual International Symposium on Code Generation and Optimization*. 9.

[9] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. [n. d.]. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages* (Pittsburgh, Pennsylvania, USA, 2015-10-21). ACM, 78–90. https://doi.org/10.1145/2816707.2816714

[10] Nguyen Xuan Hoai and Robert I. McKay. [n. d.]. *A Framework For Tree-Adjunct Grammar Guided Genetic Programming.* http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.4037&rep=rep1&type=pdf

[11] Michael Kommenda, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller. 2012. On the Architecture and Implementation of Tree-based Genetic Programming in HeuristicLab. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation* (Philadelphia, Pennsylvania, USA) *(GECCO '12)*. ACM, New York, NY, USA, 101–108. https://doi.org/10.1145/2330784.2330801

[12] Oliver Krauss. 2021. *Amaru - The Amaru Framework for Genetic Improvement and Pattern Mining in Graal and Truffle.* https://doi.org/10.5281/zenodo.6104384

[13] Oliver Krauss. 2022. *Pattern Mining and Genetic Improvement in Compilers and Interpreters.* Ph. D. Dissertation.

[14] Oliver Krauss, Hanspeter Mössenböck, and Michael Affenzeller. [n. d.]. Towards Knowledge Guided Genetic Improvement. In *2020 IEEE/ACM International Workshop on Genetic Improvement (GI)* (2020-10). https://doi.org/10.1145/3387940.3392172

[15] William B. Langdon, Westley Weimer, Justyna Petke, Erik Fredericks, Seongmin Lee, Emily Winter, Michail Basios, Myra B. Cohen, Aymeric Blot, Markus Wagner, Bobby R. Bruce, Shin Yoo, Simos Gerasimou, Oliver Krauss, Yu Huang, and Michael Gerten. [n. d.]. Genetic Improvement @ ICSE 2020. 45, 4 ([n. d.]), 24–30. https://doi.org/10.1145/3417564.3417575

[16] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. [n. d.]. A Cost Model for a Graph-Based Intermediate-Representation in a Dynamic Compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, 2018-11-04) *(VMIL '18)*. ACM, 26–35. https://doi.org/10.1145/3281287.3281290

[17] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. [n. d.]. Extended Comprehensive Study of Association Measures for Fault Localization. *Journal of Software: Evolution and Process* 26, 2 ([n. d.]), 172–219. https://doi.org/10.1002/smr.1616

[18] Daniel Manrique, Juan RÃŋos, and Alfonso RodrÃŋguez-PatÃşn. [n. d.]. *Grammar-Guided Genetic Programming.* Encyclopedia of Artificial Intelligence. https://doi.org/10.4018/978-1-59904-849-9.ch114

[19] OpenJDK. 2022. *Graal Project.* http://openjdk.java.net/projects/graal/

[20] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. [n. d.]. Snippets: Taking the High Road to a Low Level. *ACM Transactions on Architecture and Code Optimization* 12, 2 ([n. d.]), 20:20:1–20:20:25. https://doi.org/10.1145/2764907

[21] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, and Thomas Würthinger. [n. d.]. Compilation Queuing and Graph Caching for Dynamic Compilers. In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages* (New York, NY, USA, 2012-10-21) *(VMIL '12)*. ACM, 49–58. https://doi.org/10.1145/2414740.2414750

[22] David R. White. [n. d.]. GI in No Time. In *Proceedings of the Genetic and Evolutionary Computation Conference* (New York, NY, USA, 2017-07-15) *(GECCO '17)*. ACM, 1549–1550. https://doi.org/10.1145/3067695.3082515

[23] Christian Wimmer and Thomas W˜rthinger. 2012. Truffle: A Self-optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA). ACM, New York, NY, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[24] Thomas Würthinger, Christian Wimmer, Andreas WöÃ§, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. [n. d.]. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (New York, NY, USA, 2013-10-29) *(Onward! 2013)*. ACM, 187–204. https://doi.org/10.1145/2509578.2509581