LABELED MODULES IN PROGRAMS THAT EVOLVE

LABELED MODULES IN PROGRAMS THAT EVOLVE

A Dissertation Presented

by

ANIL KUMAR SAINI

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2022

College of Information and Computer Sciences

LABELED MODULES IN PROGRAMS THAT EVOLVE

A Dissertation Presented

by

ANIL KUMAR SAINI

Approved as to style and content by:

Lee Spector, Chair

David Jensen, Member

Hava Siegelmann, Member

Adam Porter, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

ACKNOWLEDGMENTS

It is often said that it takes a village to educate a child. Similarly, for the successful completion of my PhD program, I have been extremely fortunate to have a proverbial ‘village’ of people who have guided and supported me throughout the PhD program.

First and foremost, I would like to express my sincere gratitude to my PhD advisor, Lee Spector, for guiding me on this journey to becoming an independent researcher. I learned a great deal from him on how to frame research questions and tackle them with creative rigor. His insistence that science advances through both successes and failures made me appreciate research even more.

The support and supervision of the members of my dissertation committee, David Jensen, Hava Siegelmann, and Adam Porter have been invaluable in making me look at my own work through different angles and improving my dissertation in the process.

The work presented in this dissertation has been borne, at least in part, out of the numerous discussions I have had with the members of the Hampshire College Computational Intelligence Lab and the Amherst College PUSH Lab. I especially appreciate the help from Thomas Helmuth and Edward Pantridge.

Some of my most memorable moments were spent in the company of these amazing human beings: Becca, Harshika, Kaavya, Likhita, Noopur, Prerana, Rajkumar, Sreeparna, Umang, Vijeta. They were always there to listen to my problems in research and otherwise and offer sage advice. They appreciated the work I did even when at times it would have sounded gibberish to them. The summer of 2020 spent in 12A Brandywine playing board games will always have a special place in my heart. The town of Amherst with some of the most scenic hikes and cozy cafes gave me

ample opportunity to relax and rejuvenate. A special shout-out to the Montague Bookmill for being the most gorgeous place to read books and reflect on life.

My parents, Nirmala and Sumer, and my sister, Heman, have supported me throughout my PhD studies and I could feel their care and warmth even though we lived in countries several time zones apart.

And finally, this journey would have been drab and dull if not for the presence in my life of my dearest friends Pooja and Ravi. Nothing can replace the jamming to our favorite songs on numerous road trips and the daily discussions on mundane topics over chai. I consider myself fortunate to have these friends who gave me the strength and space to be a truer version of myself.

ABSTRACT

LABELED MODULES IN PROGRAMS THAT EVOLVE

SEPTEMBER 2022

ANIL KUMAR SAINI

B.Tech., INDIAN INSTITUTE OF TECHNOLOGY ROORKEE

M.S., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Lee Spector

Multiple methods have been developed for Inductive Program Synthesis, i.e., synthesizing programs consistent with a set of input-output examples. One such method is genetic programming, which searches for programs with desirable properties from the space of all possible programs through an iterated process of variation and selection that is inspired by natural evolution. Genetic programming has been successful in solving problems from multiple domains. These problems are often challenging because of the range of data types and control structures they require to be solved. Nonetheless, there are many programming problems that are routinely solved by human programmers that cannot be solved in an automated way by genetic programming or any of the other techniques presented in the program synthesis literature.

Among many other heuristics, humans use modularity to write complex software: they make use of modules in the form of classes, functions, etc. Recognizing the advantages of writing modular programs in software engineering, the need to evolve modular programs has been felt in genetic programming as well. Although there have been multiple efforts to induce modularity in evolving programs, the ability to evolve modular programs has not improved the performance for most of the systems on previously solved and unsolved problems. One of the reasons might be that although modules are advantageous in a variety of ways, they can also adversely affect evolution since the addition or deletion of modules often affects the performance of programs much more strongly than the addition or deletion of single instructions.

In this dissertation, we study the processes that make labeled modules safe for use by programs that evolve. To that end, this dissertation makes the following contributions: (a) We develop GLEAM (Genetic Learning by Extraction and Absorption of Modules), a framework to evolve modular programs in Genetic Programming. The main idea behind its design is to make modules safe for use by the evolving programs. (b) We test various hypotheses for why modules are useful for the evolving programs. (c) We propose two metrics to measure code reuse and repetition in the evolving programs.

We run experiments in a genetic programming system called PushGP, which evolves programs in a stack-based programming language called Push.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

More often than not, writing what a function does is easier than writing the function itself [36]. That is where the active area of research in inductive program synthesis comes in; it attempts to produce a program that is consistent with a given set of input-output examples. We just need to know *what* a program should do, and the system figures out *how* it should do it.

Multiple techniques that have been employed to synthesize programs cover enumerative search, constraint solving, stochastic search, deduction-based programming by examples, etc [17]. Genetic Programming is one of the approaches in stochastic search, which employs the processes of variation and selection inspired by natural evolution to search for the program with desirable properties from the space of all possible programs. It has been successful in finding programs that solve difficult problems in certain specialized domains, including but not limited to antenna design, electrical and quantum circuit design, symbolic regression, etc., and has produced solutions that perform better than or as well as the solutions proposed by humans [28]. It has also been successful in finding programs that solve exercises from introductory programming textbooks, which present challenges because of the range of data and control structures they require [23]. Nonetheless, there are many kinds of programming problems that are routinely solved by human programmers that cannot be solved in an automated way by genetic programming or any of the other techniques presented in the literature.

One of the heuristics that can help scale up the problem-solving capability of genetic programming systems is modularity. The inspiration comes from how human programmers use modularity to write complex software; they decompose the problem into multiple parts and use classes, functions, and other programming constructs to solve them. Recognizing the importance of modularity in writing programs in software engineering, there have been multiple efforts to evolve modular programs in genetic programming as well. However, the ability to evolve modular programs has not improved the performance for most of the systems on previously solved and unsolved problems. One of the reasons might be that although modules are advantageous in a variety of ways—for example, they allow the programs to reuse code segments—they may not be 'safe' for use for evolution in many ways. First, the addition or deletion of modules affects the performance of programs much more strongly than the addition or deletion of single instructions since modules often contain multiple instructions. Second, if a module is being used at multiple locations in a program, any change made to the module in one location will affect all other locations as well.

In this dissertation, we attempt to study the processes that make labeled modules safe for use by programs that evolve. To that end, this dissertation makes the following contributions:

1. We develop GLEAM (Genetic Learning by Extraction and Absorption of Modules), a framework to evolve modular programs in Genetic Programming. The main idea behind its design is to make modules safe for use by the evolving programs.

2. We propose a set of hypotheses to explain why modules seem to be useful for the evolving programs in the genetic programming systems that evolve modular programs. We test them in the context of GLEAM. The hypotheses are listed below:

- Programs use labeled modules to reuse code within the same program.
- Programs use modules to reuse code from other programs.
- Programs use modules to protect certain code segments from changing too frequently.
- Modules are used by the evolving programs to organize information in hierarchical or even more complex ways; programs can call modules, and modules can call each other as well as themselves.

3. We propose two metrics that measure code reuse and repetition in evolving programs and provide some information about their structure and complexity.

1.1 Genetic Programming

In genetic programming, a population of programs is modified based on their fitness (errors on a set of test cases) until a program that satisfies all the test cases is found or a pre-defined number of iterations (called generations) are passed. The initial population contains randomly generated programs. After that, going from one generation to the next involves selecting *promising* (though not always the best) individuals from the current generation that can serve as parents for the next one, and then using these selected parents as inputs to genetic operators that produce the offspring or the population for the next generation. The most common genetic operators are:

- **Crossover:** Two parent programs are combined, often by taking parts from each of them, to form a new child program.
- **Mutation:** A parent program is altered by adding new parts or deleting or replacing existing parts to create a new child program.

To conduct various experiments for the purpose of this dissertation, we use a genetic programming system called PushGP [57, 54] which evolves programs in a stack-based programming language called Push. There are many reasons for this choice:

- PushGP supports multiple data types (integers, floats, strings, vectors, etc.) and programming constructs (conditionals, loops, etc.) in programs. In fact, in Push, every data type has a separate stack. Instructions in Push can take their inputs from and place their outputs on different stacks. A program is basically a list, possibly nested, of instructions and literals. All these features make it possible to build modules containing any number and type of instructions.

- It has produced state-of-the-art results in program synthesis [50], and therefore, we can test whether the addition of modularity can further improve the performance of PushGP.

- There have been prior efforts in evolving modular programs in PushGP [56, 53, 52] which were successful in a limited number of settings. Our work, as presented in this dissertation, partially builds on these concepts.

1.2 Other Program Synthesis methods

Although there are multiple other methods to synthesize programs such as FlashFill [16], DeepCoder [2], DreamCoder [14], providing a detailed description of them and comparing them with genetic programming would be out of scope for this dissertation. Nonetheless, we briefly discuss one popular technique here.

The program synthesis methods, based on Large Language Models (LLMs), use generative deep neural networks that are trained on natural language text and publicly available source code from websites like Github. When given a query in the form of text or a partially written program, it generates the corresponding program as output. The examples of such systems include Codex [8], Github Copilot, Amazon

CodeWhisperer, etc. Most of these networks build on a natural language processing model called GPT-3 [5].

One advantage of using these models is that once they are trained on a corpus of source code, they can generate any number of programs. However, they have many drawbacks as well. First, they require a large amount of training data and compute to train. Second, if they are trained on one domain (programs written in one programming language, for example), generating programs from a different domain might require a substantial amount of retraining of the models. And the absence of training training examples for a certain novel domain may make things even more difficult. Third, although they generate syntactically correct programs, there is no guarantee of behavioral correctness, i.e., the programs generated by them may not satisfy the test cases the user might test them on.

Compared to LLMs-based program synthesis, genetic programming does not require a large amount of training data and can work well with a reasonable number (typically hundreds) of test cases. It can also generate programs in a domain where there are no programs available as training data.

1.3 Modularity

Depending on the field of study, there can be different definitions of what exactly a module is. For the purpose of this dissertation, however, a **module** is defined as a code segment that can be called by a program using some unique label or identifier. This is similar to how functions are called by programs in regular programming languages using their names as identifiers. Hence, if a program has **modularity**, it implies that it has access to a library of labeled modules.

1.4 Contribution 1

1.4.1 Making Modules Safe for Evolving Programs

Modules can be useful in a variety of ways: they can be reused by the program, they can be changed independently of the program that uses them, they can make references to each other and organize information in a hierarchical fashion, etc. But, modules can also pose some great difficulties for evolving programs: the addition or deletion of a reference to a module can affect the behavior of a program much more significantly than the addition or deletion of a single instruction, and using modules makes the evolution more unstable as changes made to a module at one place will affect all the places that module is being used.

To mitigate the above-mentioned negative effects, we propose the following ways to make modules *safer* for evolving programs:

Incorporation by Reference. Instead of directly inserting the code contained in modules into programs, there should be a mechanism to insert the references to such modules instead. This makes it easy to remove such modules from programs through mutation operators in case the module has undesirable effects on the programs.

Stability of Reference. When applying genetic operators on parent programs to produce child programs, the modules which are being used by those programs should change only slightly using, for example, a very low rate of mutation. The modules which are not being used by the programs can be changed arbitrarily.

Local Scopes for Modules. Similar to how functions are used in general programming languages, the operations performed inside the modules should not affect the variables used by the rest of the program.

1.4.2 GLEAM

The above-mentioned principles of making modules safe for use have been used in designing a framework for evolving modular programs called Genetic Learning by

Extraction and Absorption of Modules or GLEAM [45, 47]. We implement GLEAM in PushGP. Under this framework, every individual, in addition to the program that is executed to get errors, also has a local library of modules. The program can call these modules using special referencing instructions. Modules can call themselves as well as each other. After every generation, in addition to the usual genetic operators, three other operators are applied to every individual to update its library: Extraction, Absorption, and Mutation for modules.

Extraction: During extraction, modules in the local library that are not being used by the program are replaced with new modules with some probability. Typically, the unused modules are replaced by code segments from the same program.

Absorption: When applying the absorption operator on the individual, every module reference in the program, as well as other modules, can get 'absorbed' with some probability. The 'absorption' happens when a module reference (in the program or other modules) is replaced by the code segment it refers to.

Mutation for modules: In addition to applying the mutation operator on the individual's program, the modules in its library may also be mutated.

When the GLEAM framework is used with PushGP, it leads to significant improvement in the number of successes for various benchmark software synthesis problems. The details of these experiments are given in later chapters.

1.5 Contribution 2: Why are Modules Useful?

We test various hypotheses for why modules are useful for the evolving programs using the GLEAM framework. They have been mentioned in the previous work (e.g., Koza [30]) in different forms but have not been explicitly tested. Description of some of these hypotheses and the results obtained using experiments to test them are listed here:

1. Programs use labeled modules to reuse code segments within the same program.

   Results: There seems to be evidence against this hypothesis as, on average, every module is used only once in a given program. Additionally, when programs are restricted to use every module only once, there is no significant difference in the performance on the benchmark problems.

2. Programs use labeled modules to reuse code between programs.

   Results: There seems to be evidence for this hypothesis as when the system does not allow programs to get modules from other programs into their own library during extraction, the performance deteriorates.

3. Programs use labeled modules to 'protect' code segments from changing too frequently.

   Results: This also does not seem to hold as we tried different mutation rates for modules, and there does not seem to be any significant difference in the performance.

4. Programs use modules to organize information in a hierarchical way.

   Results. This does not seem to hold, as the performance of the system does not deteriorate significantly under the condition where modules are not allowed to call each other or themselves.

1.6 Contribution 3: Code Use Metrics

In software engineering, there are many metrics that measure different aspects of code use: Lines of Code (LOC), Coupling, Cohesion, etc. In genetic programming, we introduce two metrics: Reuse and Repetition. **Reuse** measures how frequently a copy of a module gets executed. **Repetition**, on the other hand, measures how frequently the code of a given module appears in the program. In Reuse, one copy gets executed multiple times, whereas, in Repetition, there are already multiple copies present in the program, each of which gets executed once.

These metrics provide information about the importance of some code segments relative to others, and that information might be used in various processes related to modularity. For example, if some segment is used frequently, it is probably very important to the program and might be used to create new modules for the program.

We also study how parent selection algorithms influence code reuse in the population of evolving programs. From various experiments, we observe that the number of individual programs with non-zero reuse (i.e., the programs with at least one code segment that is used multiple times) in the population is highly correlated with the success rates for different selection methods. For example, lexicase parent selection algorithm, which has the highest success rate among the selection methods considered, also leads to the highest number of individuals with non-zero reuse.

1.7 Collaborators

This dissertation contains excerpts from various publications where the work was done in collaboration with Lee Spector [29, 30, 27, 25, 26] and Thomas Helmuth [48].

1.8 Supporting Grants and Systems

This material is based upon work supported by the National Science Foundation under Grant No. 1617087. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This work was performed in part using high performance computing equipment, specifically the Swarm cluster, obtained under a grant from the Collaborative R&D Fund managed by the Massachusetts Technology Collaborative.

Some work was also performed on Hampshire College's high-performance computing cluster called Fly.

CHAPTER 2
MODULARITY

The terms "modularity" and "module" often have different definitions for different fields of study. In its most common understanding, however, *modularity* can be defined as the property of being organized into decomposable units, also known as *modules*. Consequently, which parts are considered as modules is determined by the kind of *organization*, and the level of *decomposability*—the interactions between and within modules—allowed by a particular definition.

As alluded to in the previous chapter, for the purpose of this dissertation, we will consider a module to be a code segment that can be called by a given program using some unique label or identifier. But to give a broader picture of the usage of the concept of modularity, in this chapter, we will present an overview of modularity as defined and used in various domains, with a particular focus on how it has been implemented in genetic programming systems.

Although there are multiple ways of studying modularity [7], in this dissertation, we will focus on its 'structural' aspects. In other words, we will study how a given system is composed of different structural units.

2.1 Modularity in non-genetic programming systems

In software engineering, a 'modular' design in a program or software usually refers to its decomposition into different parts such as classes, structures, functions, files, etc. Although, in this chapter and elsewhere, the terms 'module' and 'function' may be used interchangeably, we will reserve 'functions' to mean the formal programming constructs as used in regular programming languages.

In biology, there are multiple definitions of modularity, and they can measure entirely different phenomena depending on the context in which they are used—whether we are working at the level of an individual or a population, whether we are considering developmental (e.g., traits in an organism which develop together) or functional modules (e.g., different parts of the brain that perform slightly different functions), etc [61]. Nevertheless, a somewhat consistent way to define a module may be to consider it as a group of elements acting as a semi-autonomous entity that can evolve and function relatively independently from other modules [15].

Moreover, in certain networks, whether or not they model some biological processes such as gene regulation, modules can be identified as the highly connected clusters of nodes that are sparsely connected to nodes in other clusters [9].

As far as the methods for program synthesis are concerned, the idea of using a library of modules to aid the synthesis of programs is not unique to genetic programming. DreamCoder [14], for example, uses a version of wake-sleep learning whereby programs are synthesized by a neural network in the 'wake' phase that uses the concepts or procedures which were learned in the 'sleep' phase. The library containing the *concepts*, which are basically labeled modules, is updated by extracting segments from the programs already synthesized by the neural network. Although DreamCoder can synthesize programs from various domains, it is not clear whether it can solve standalone problems well without the availability of similar problems from the same domain. Its ability to synthesize complex software requiring the use of multiple data types in inputs, outputs, and other variables also remains to be seen.

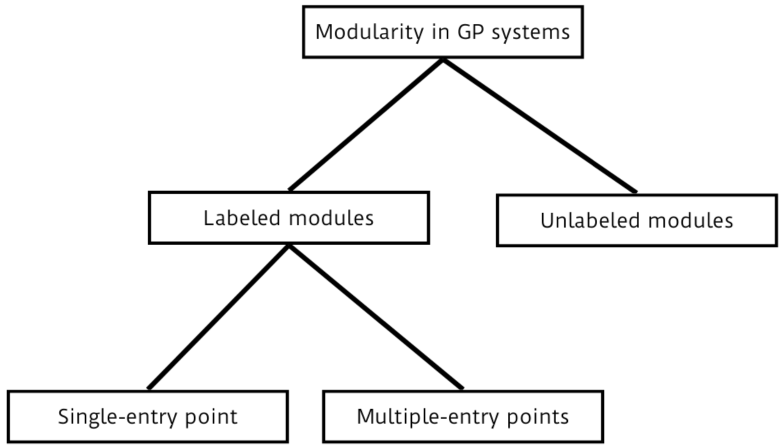


Figure 2.1: Classification tree of the genetic programming systems that attempt to evolve modular programs.

2.2 Modularity in genetic programming systems

In this section, we discuss the concept of modularity as understood in the field of evolutionary computation in general and genetic programming in particular. Modularity in this setting essentially means the presence of modules in the evolving entity, typically a program. A module, in turn, is part of the evolving program exhibiting some sort of independence from other parts [31]. To make the discussion about modularity in genetic programming systems more understandable, we will use the categorization as given in Figure 2.1. We will discuss each of these categories in detail in the following sections.

Depending on how they are used by the evolving programs, modules can be of two types: labeled and unlabeled. Whereas labeled modules have identifiers or labels which are used by the programs to call them, unlabeled modules exist as different parts of the program which are executed in a particular order. The equivalent of labeled modules in human programming would be functions, procedures, classes, etc., and unlabeled modules would be loops, conditionals, or any such set of instructions that are grouped in such a way that they execute together.

2.2.1 Labeled Modules

This section deals with the frameworks where evolving individuals have access to *labeled* modules. These modularity-inducing systems can be further grouped into two categories. In the systems in the first category, which we call **Single-entry-point systems**, an evolving individual is composed of a 'main' program and a set of modules. These modules can be local to an individual or shared by all individuals in the population. Execution always starts from the main program, which can call other modules during execution. Modules can also call each other. In the second category, which we call **Multiple-entry-points systems**, an evolving individual is made up of modules. There is no concept of a main program as such, and the execution can start from any of the modules based on the signal from the environment.

Single-entry-point Systems. One of the very first uses of modules in genetic programming can be traced back to John Koza's Automatically Defined Functions (ADFs) [30] in his book in 1992. An evolving individual is basically a tree with one or more ADF branches and one result-producing branch. Only the result-producing branch is executed to calculate the output, but this branch can call ADFs at other branches. ADFs were shown to improve the probability of success on a number of problems, such as parity functions. In its initial formulation, ADFs were not flexible enough in the sense that the form of the functions, including their name, arguments, etc., had to be defined in advance. To remedy this, various modifications were proposed, including but not limited to, Architecture Altering Operations [29], Automatically Defined Macros [51], etc.

Another method of building and using modules in trees is Module Acquisition [1] which involves the processes of compression and expansion. In compression, a subtree of an individual tree is replaced by a new function which is defined by the subtree that is being replaced. In expansion, the function call is replaced by the subtree it refers to.

Grammatical Evolution (GE) using modules is another technique in this category. First, all the subtrees in the individuals in the population are assessed on their usefulness in their respective program trees, and the better-performing ones are considered to be modules. An example measure of usefulness can be the difference in the fitness of the whole tree and the tree with the subtree replaced by a randomly generated tree. This whole process is called *module identification* [58]. Next, the underlying grammar is modified by adding additional production rules so that individuals in future generations are able to use such modules.

In Embedded Cartesian Genetic Programming (ECGP) [59], modules are created by a process called compression in which a section of the genotype is made into a module, and that section is replaced by a reference to that module. The inputs and outputs of the module are defined in a way such that the compression operator does not change the final output of the program.

Tag-based modules [56, 52], whereby programs can label code segments with integer tags and later refer to them during execution, have been used in PushGP and elsewhere. One of the main features of these tags is 'inexact matching' which can be described in the following way. The modules are indexed by tags. If a module call contains a tag that is present in the set of available modules, the corresponding module is returned. Otherwise, the module with the closest tag is returned instead of throwing an error. These tagged modules have shown limited success in problems like the lawnmower problem and the obstacle-avoiding robot problem.

Some work has also been done to use a set of modules between different runs for a single problem. For example, a Run Transferable Library [25], which is a collection of functions known as Tag Addressable Functions (TAFs), is used and updated in one run and transferred to the next one for a certain number of runs for the same or similar problems. All the individuals in the population are free to use the modules in this global library.

Multiple-entry-points Systems. In SignalGP [33, 34], a program is a set of functions that can be accessed by their identifiers called tags. Events in the environment also contain tags and can trigger functions with appropriate tags in the program. Tangled program graphs (TPG) [26] use two distinct populations, one for teams and the other for programs. A team is a collection of pointers to programs, which are executed to calculate the fitness of the team. Here, teams act as individuals and programs as modules.

As we will discuss in the later chapters, the modularity-inducing framework presented in this dissertation, i.e., GLEAM, and other related topics, use the single-entry-point framework.

2.2.2 Unlabeled Modules

A given code chunk of an evolving program may be considered a module if all the instructions in that chunk execute together and it has some sort of independent existence. For example, since all the instructions in the body of a loop execute together for a certain number of iterations, the body of the loop can be considered a module.

PushGP has a built-in mechanism for the evolution of unlabeled modules through code self-manipulation [56]. Basically, there are instructions in Push that can manipulate the code on `exec` stack as it runs. Some examples include the instructions that can copy a code chunk a certain number of times, make groups of instructions execute in a different order than the one they are written in the program, etc.

2.2.3 Metrics to measure modularity

Although there are multiple definitions of modularity in different fields, there are only a few metrics that measure it. In software engineering, *coupling* measures the interdependence between different modules, and *cohesion* measures the amount of interaction among components of a particular module [12]. A modular software design, therefore, is characterized by low coupling and high cohesion.

In genetic programming, modularity can be measured in multiple ways. For example, a metric called Q-metric [38] can be used for programs that can be represented as networks. It measures the interactions within and between groups of nodes. Other metrics that try to measure modularity in evolving programs are Reuse and Repetition [41], which will be discussed in detail in Chapter 7. Reuse measures how frequently a copy of a module gets executed. Repetition, on the other hand, measures how frequently a given module appears in the program.

In this dissertation, we will study the type of modularity whereby an evolving individual contains a 'main' program and a set of 'modules'. Every module is 'labeled' in the sense that it has an identifier that the program uses to call it during execution. It can be a string name, integer label, or even a production rule (as in Grammatical Evolution). The modularity-inducing frameworks in Section 2.2.1 fall in this category.

CHAPTER 3
PUSHGP AND BENCHMARKS

To conduct various experiments for the purpose of this dissertation, we use a genetic programming system called PushGP [57, 54], which evolves programs in a stack-based programming language called Push. Specifically, we use the PushGP system written in Clojure called Clojush[1]. In this chapter, we will give a brief overview of the PushGP system and the benchmark problems that we use in our experiments.

3.1 PushGP

In Push, every data type has a separate stack. Instructions in Push can take their inputs from and place their outputs on different stacks. A Push program is basically a list, possibly nested, of instructions and literals. While instructions, when executed, perform certain computations on the input data to produce output to be placed on a particular stack, literals are simply placed on the appropriate stacks during execution.

Whenever a program is executed, it is first put onto a special stack called the `exec` stack. After that, in each execution step, the top item on the `exec` stack is executed. If the top item is a list, it is 'unpacked,' i.e., the contents are placed on the exec stack in such a manner that the first element in the unpacked list is on the top of the exec stack after the unpacking. If the top item is a constant or instruction, it is handled as mentioned before.

Instructions come in different forms: simple functions that act on their inputs and execute only once, conditionals that can execute one set of instructions or the other

[1]https://github.com/lspector/Clojush/

depending on the values of other inputs, looping instructions which can execute a set of instructions multiple times, etc.

Let us take a simple example to understand PushGP better: `(2 3 :integer_mult 'abc' :string_reverse)`. When the execution starts, the whole program is put onto `exec` stack. Now, since the top item is a list, it is unpacked such that 2 is on top and `:string_reverse` is at the bottom. After that, since 2 is now on top, it is executed, which basically means putting it on `integer` stack. After that, the top item is 3, which is also put onto the `integer` stack. Then, the instruction `:integer_mult` is executed, which removes the top two items on `integer` stack (which are 2 and 3), multiplies them and puts the result back to the `integer` stack (the top integer is now 6). Now, the next item on the `exec` stack is "abc", which is put onto the `string` stack. The last intruction `:string_reverse` removes the top item from the `string` stack ("abc"), and puts the reverse of that string ("cba") back onto the `string` stack.

Push supports most of the programming constructs present in regular programming languages, such as conditionals, loops, recursion, etc. Sometimes, the Push programs can have infinite recursion and loops. In order to prevent that, PushGP sets some upper limit on the number of instruction evaluations that can be performed while executing a given program. Usually, this limit is set by the user in advance and is different for different problems. If this limit is exceeded, the program terminates, and the remaining instructions are not executed. However, the program may nonetheless produce outputs and may be considered to be a valid program even if it was terminated for exceeding this limit.

3.2 Benchmarks

Although PushGP can be used to solve a variety of problems, including the ones solved by any other genetic programming system, in the experiments in this dissertation, we work with problems from two sets of General Program Synthesis Benchmark

Suite, which we call PSB1 [23] and PSB2 [19]. Various features of the problems in these benchmark suites are:

1. They are taken from the various introductory programming textbooks, online competitions, etc.

2. They use a variety of input and output data types such as integers, strings, lists, boolean variables, etc.

3. The problems are of varying levels of difficulty. In fact, some of the problems have not been solved by any automated system in the literature, as far as we are aware.

Note that although we use PushGP on a set of benchmark problems in our experiments in this dissertation, the conclusions we draw from the results should be applicable, with some possible modifications, to most of the genetic programming systems as long as those systems also have some of the advanced programming constructs used by PushGP such as conditionals, loops, etc.

CHAPTER 4

MAKING MODULES SAFE FOR EVOLUTION

Modules, in the form of code segments with some identifier that can be called by the evolving programs, can be useful in a variety of ways:

1. A module can be called multiple times by the program. Hence, if a code segment needs to be executed multiple times at possibly different locations, the evolution does not need to build the code segments from scratch at those locations; it can just refer to the same code segment again and again by referencing the appropriate module.

2. The modules can be changed independently of the program that uses them.

3. The modules can make references to each other and organize information in a hierarchical fashion, which is difficult, if not impossible, to achieve without using any modules.

4. The modules make it easy to transfer information between programs. Instead of exchanging individual instructions, say, during crossover, the whole modules can be exchanged between them.

But, in addition to the various advantages listed above, modules can also pose some great difficulties, especially when they contain arbitrary code.

1. The addition or deletion of a reference to a module can affect the behavior of a program much more significantly than the addition or deletion of a single instruction.

2. Any changes made to a module can affect the performance of the program much more strongly than changes made to any of the instructions. Take, for example, a module that is called at multiple places in the program instead of there being copies of the same code segment at those places. When there are multiple copies, changing one copy does not affect other copies. But with using the module, whenever you want to make changes at one place, the code at other places also changes.

To lessen the adverse impacts of using modules, we provide a set of guidelines that can make modules containing arbitrary instructions safe for evolving programs to use. We use these guidelines in designing the modularity-inducing system called GLEAM, as discussed in the next chapter.

4.1 Incorporation by Reference

Sometimes, during the course of evolution, some instructions might be added to the evolving program that have adverse effects on the performance of the program, i.e., they increase the errors of the program on a set of test cases. To rectify that, the mutation operators are designed in such a way that there is a non-zero probability of such instructions being removed or replaced in the following generations.

Similarly, modules, which are made up of arbitrary instructions, can sometimes have undesirable behavior as well. This issue is compounded by the fact that modules contain multiple instructions, and executing a module effectively means executing all the instructions at once. To prevent that, similar to the case with simple instructions, the reference mechanism should be such that the references can be removed through some mutation operators.

4.2 Stability of Reference

For simple instructions, the definitions describing their function remain fixed throughout evolution, but for modules, their definitions can change. In fact, the whole code segment, a given reference points to, may be replaced by an entirely new code segment. In order to prevent drastic changes like this, the following strategy can be implemented. Modules that are being used by the program should be changed slightly if at all, using a lower rate of variation (mutation, crossover, etc.) than that used for the program. Modules that are not being used by the program can be changed arbitrarily, however.

4.3 Executing modules in their own environments

Although functions in most other programming languages can return multiple values and change some variables used by the program (for example, lists passed by reference) that calls them, they nonetheless do not change large parts of the global scope of the program. Similarly, the operations performed inside the modules should not affect the variables used by the rest of the evolving program and ideally should return a single output value to the program.

CHAPTER 5

GLEAM: GENETIC LEARNING BY EXTRACTION AND ABSORPTION OF MODULES

In this chapter, we present and analyze a new framework for the evolution of modularity called GLEAM (Genetic Learning by Extraction and Absorption of Modules), whereby evolving programs can create and use modules during evolution. GLEAM leads to significant improvement in the success rate for multiple benchmark problems.

Additionally, GLEAM's flexible architecture and tunable parameters allow researchers to test different methods related to the generation, propagation, and use of modules in genetic programming.

5.1 Introduction

Genetic programming systems have been shown to be capable of synthesizing programs that make use of multiple data types, conditionals, loops, and other control structures that, for human programmers, support the development of complex programs. They can successfully solve many problems from the introductory programming textbooks [23]. To solve more complex problems, however, like many that are routinely solved by human programmers, additional breakthroughs may be required with respect to the evolution of modularity. Despite multiple efforts in this direction, achieving scalability through modularity remains one of the open issues in the field of genetic programming [39].

Many of the techniques proposed to encourage modularity in evolving programs are designed to work only for a handful of modules, and it is not clear that these techniques are able to produce large-scale software that involves dozens or more modules.

In this chapter, we present and analyze a new framework called GLEAM (Genetic Learning by Extraction and Absorption of Modules), in which an evolving program can make use of a local library of modules that is propagated from generation to generation (with possible variation) along with the program's code. Programs can call modules, and modules can also call each other. Figure 5.1 illustrates a typical evolving individual in GLEAM.

The framework itself does not specify how module arguments and return values and their types are handled. Details of argument handling depend on the underlying genetic programming system; below, we spell out the way in which this is handled in our own experiments.

After every generation, the library is updated in the following way: modules that are not being used by the individual may be replaced by modules extracted from the individual itself or elsewhere, modules may be mutated, and module references may be replaced by their corresponding code segments (which we call "absorption" of the module). New modules are typically extracted from the main program or other modules associated with the same program, but they may also be taken from other individuals or any other external source.

Below, we present the GLEAM framework in more detail and describe the results of experiments with GLEAM on benchmark software synthesis problems. We analyze the effectiveness of GLEAM for evolving modular programs and the improvement in software synthesis performance relative to experiments in which GLEAM is not used. We also briefly describe additional experiments that illustrate the use of GLEAM as a platform for testing different methods for evolving and using modules in genetic programming.

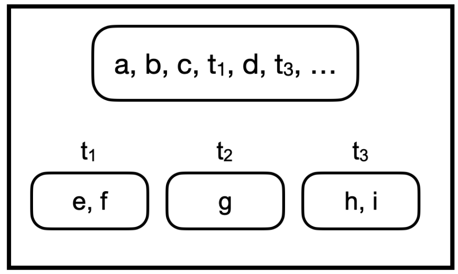


Figure 5.1: An individual in GLEAM. The first part shows the program and the second part shows the modules labeled with tags. The letters a, b, c, etc. denote regular instructions, whereas t_1, t_2, etc., denote module references.

5.2 GLEAM

In this section, we describe various aspects of the GLEAM framework. Although the description has been kept as general as possible, some details on how GLEAM has been implemented in our experiments are also given.

Each individual in a GLEAM population has a main program and a local library containing a number of modules. Only the main program is executed for error testing. Modules are not executed directly, but called by the main program or by other modules during execution. Modules are identified and called using integer tags.

Libraries are updated each generation, ideally in a way that will cause useful modules to be retained and useless modules to be replaced by new ones over the course of evolution. We define the usefulness of modules in a simple way: those modules which are being referenced by the program directly or indirectly are considered useful. Those that cannot be reached by chains of references starting in the main program are considered useless.

GLEAM is a general-purpose technique; it can be implemented with most, if not all, of the existing genetic programming systems. The only requirement for the underlying system is that the modules used by the evolving individuals in the system have *labels* that can be used to call them. For the sake of concreteness, the description of GLEAM in this section assumes the use of linear genome representations, but we provide some suggestions for implementation in tree-based genetic programming systems as well.

5.2.1 Initializing the library

Modules are initialized in the same way as the program itself, using the same instructions that are available to the program. The size of initial modules, however, is usually kept smaller than the size of initial programs.

The number of modules in the library of an individual can be set to any size. In the version of GLEAM that we use for the experiments described in this chapter, each individual library contains a small fixed number of modules (10). However, we consider this to be a special case of the more general GLEAM architecture, in which the number of modules may be increased over evolutionary time, with a specified number of modules being added every time a specified number of generations passes. More research is required to determine the effects of various schedules for increasing the number of modules and to understand the behavior of the system with much larger numbers of modules.

Within the context of a fixed, small number of modules, we found in exploratory experiments that the precise number makes little difference, and we picked 10 as the number of modules for the experiments here simply because it is a round number.

5.2.2 Referencing the modules

Whenever a module is called, it is provided the current state of the calling program. For Push programs, the 'state' of the system at any given point of time is the content present on various data stacks. During execution, the module can change the given state, and that state, later on, gets returned to the calling program after the module has been executed. Restrictions on the amount and type of information passed to the modules, and returned from the module calls, could be implemented by the underlying genetic programming system. In the case of PushGP, we do place some restrictions on the outputs returned by a module, which will be discussed in Section 5.3.

Calling a module requires the use of special referencing instructions. For the experiments described here, referencing instructions are implemented in the following way. In order to refer to a particular module with identifier i (also called its tag), the program uses the instruction `tagged_i`. This is similar to the procedure adopted in Spector *et al* [56]. To generate tag references in the program, a special function `tagged_erc_limit` is used. It inserts `tagged_j` in the program, where j is an integer chosen randomly between 0 and a pre-defined limit. Since the number of modules in a given library is 10, this limit is also set to 10. Based on some preliminary experiments, we found that using two of these tag-reference generators in our set of instructions available for the programs and modules seems to give better results than using other numbers. Effectively, during initialization and application of other genetic operators like mutation, referencing instructions will be chosen two times as frequently as any other specific instruction.

5.2.3 Updating the library

In Algorithm 1, we present the variation operators that are applied to an individual in GLEAM. In addition to applying the mutation operator on the program, we also update the library containing modules for every individual. In the algorithm, the `ind` variable is a data structure (dictionary, hashmap, or something similar) containing the main program and a set of modules called the library.

For every generation, whenever a new child genome is created, while applying genetic operators on it, its local library is also updated. Other details of this updating procedure for the library are given in the following subsections.

Algorithm 1: Variation operators in Genetic Programming Algorithm modified to accommodate the steps needed for updating the library in GLEAM.

```
/* Update the library                                                    */
ind['library'] := Mutate(ind['library'], mutation_rate_for_module) // Apply
    mutation on the modules
ind['library'] := Extract(ind, extraction_rate) // Replace the unused
    modules with the new ones
ind := Absorb(ind, absorption_rate) // Absorb modules in the main
    program as well as other modules

ind['main program'] := Mutate(ind['main program'],
  mutation_rate_for_program)
```

5.2.3.1 Crossover and Mutation

Although the crossover operator is not shown in Algorithm 1, it can be applied to individuals with GLEAM, for which the following additional steps might be necessary. If a part being received by the child program contains tag references, the child also receives the associated modules. In case various parts from different parents contain the same tag referring to different modules, some of those tags may be mutated to accommodate all the modules being referred to by the tags. For example, assume the child genome receives two parts, and both of them contain `tagged_2`. In order to retain modules from both parts, one of the tags can be changed to some other tag not currently in use. If no other free tag is available, the module with tag 2 from one of the parts may be dropped.

The mutation methods for the program and the modules in the library can be different, but to maintain uniformity and simplicity, we will assume the same method is used for both parts. The rates of mutation, however, might be different for the main program and modules in the following ways:

1. The mutation rate for modules is less than that of the program. In extreme cases, the mutation rate can also be zero. That would essentially mean that mutation is not applied to the modules.

2. The mutation rate for modules is the same as that of the main program.

3. The mutation rate for modules is more than that of the main program.

This difference in mutation rates can affect the dynamics of evolution. In the first case, for example, lower mutation rates essentially mean the modules are being 'shielded' from frequent mutations so that they can evolve code segments which might be too useful for the programs to be changing frequently. This is similar to the idea of multiple levels of evolution in hierarchical evolution [3], where modules in the lower layers evolve at a slower speed than the modules or programs in the upper levels. Keeping the mutation rate for modules the same as or more than that of the program would not provide any such protection to the modules.

Note that although we discussed crossover and mutation operators in detail here to show the generality of GLEAM, in our experiments conducted in this chapter, we only use the mutation operator. The reasons for doing this will be discussed later in the chapter.

5.2.3.2 Extraction

We consider a module being *used* if it is reachable by the program, i.e., if the program contains a reference to it, or if the program contains a reference to some other module which in turn contains the reference to it, and so on. In the extraction operator, every unused module gets replaced by a new module with a certain probability. Figure 5.2 gives an example of an unused module being replaced by a code segment from the program. In the figure, the module with tag t_2 is not being used by the program. During the extraction operation, the contents of this module are replaced by a sequence of instructions taken from the program itself.

Where do the new modules come from? Various methods [3, 25, 58] find the new modules using a process known as Module Search or Module Identification. In this process, developed for tree-based genetic programming, different subtrees from individuals in the population are assigned fitness depending on their usefulness in the original trees, and the subtrees with better fitness than the others are chosen as new modules. The fitness of a subtree can be assessed in various ways: difference in the fitness of the individual tree with and without the subtree, difference in the fitness with the given subtree and a randomly generated subtree inserted in its place, etc. For GLEAM, new modules may come from these places:

1. Generate a random code segment of a certain length.
2. Choose a random code segment from the current program or any other program in the population.
3. Choose an existing module from the library of the current program or any other program in the population.
4. Any other external repository. This is similar to how functions or modules are imported from external libraries when writing programs in regular programming languages.

For the first two options, the lengths of new modules would also need to be determined. This can be done by sampling lengths from a discrete distribution such as Poisson distribution, Negative Binomial distribution, etc. The above-mentioned procedures for new modules are for linear genomes. For tree-based representations, a subtree of the program can serve as the module, similar to what is done in Run-transferable libraries [25] or Hierarchical Genetic Programming [3].

5.2.3.3 Absorption

In the absorption operator, with a certain probability, every module reference in the program, as well as the modules, can get replaced by the code segment it refers to. Figure 5.3 provides an example of a module getting absorbed by the program. The tag reference t_1 in the program is replaced by the code segment it refers to, i.e.,

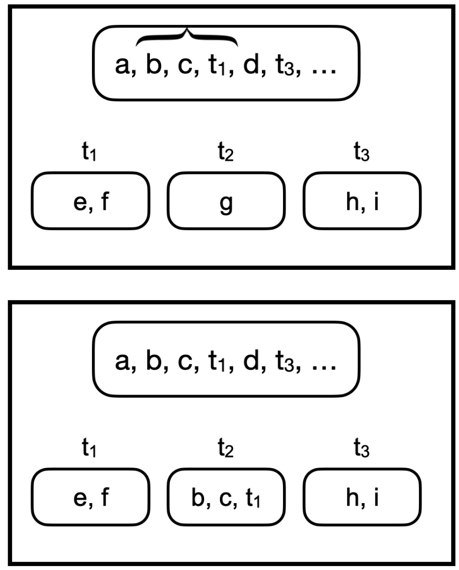


Figure 5.2: An individual in GLEAM before and after the extraction operation. Since the tag t_2 is not being used, it is replaced with the segment bct_1 extracted from the program.

'e, f.' Note that the module and the associated tag remain in the library as they were before.

One reason why expanding modules might be useful for the programs is that it frees up the tag unless the same tag is referenced somewhere else in the program. The freed-up tag-module association becomes unused and can be replaced by a new module. Another reason might be that once a module is expanded into the program, the code it contains can interact more freely with other instructions in the program than was possible before. For example, suppose there is a looping instruction in the module. Without absorption, the length of the body of that loop is mostly limited by what is present in the module (although mutation can sometimes increase the size of the module, but not by much). With absorption, that looping instruction can go to the program and include other bigger parts of the program in the loop body, possibly over many generations.

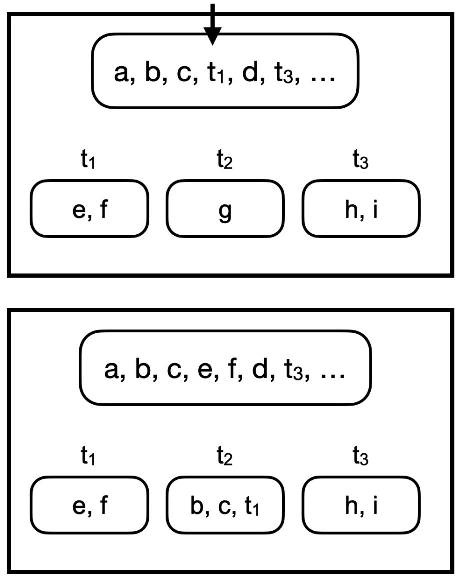


Figure 5.3: An individual in GLEAM before and after the absorption operation. The tag t_1 gets expanded in the program.

5.3 Environments

Scoped environments are used in various programming languages to provide contexts within which a function's code can bind values and perform computations with them, without affecting bindings used by the rest of the program. The PushGP genetic programming system allows evolved programs to label sequences of instructions as modules. However, those modules can also modify bindings used by the rest of the program. This increases module coupling and might therefore make modular programs less evolvable than they otherwise could be. To rectify this and to ensure that modules return single values, we implement scoped environments in PushGP, using a method that allows for the dynamic definition of arbitrary modules in a multi-type setting [48].

Instructions in PushGP take their arguments from and place their outputs on stacks of different data types. These instructions usually modify just one or a small number of stacks, usually consuming and/or producing just one or a small number of values at the tops of the stacks. By contrast, tagged modules, which may contain many instructions, are likely to produce many more changes to the states of the stacks, which serve as the primary value store for the program being executed. This differs both from the behavior of most built-in Push instructions, and also from the behavior of most functions and other kinds of modules written by human programmers, at least if they are following the recommended software engineering practices. It also seems likely to interfere with the evolution of tag-based modules since the introduction or elimination of a module is likely to have large-scale, potentially catastrophic impacts on the behavior of the program. To address this issue, we have developed an approach for limiting the effects that a module call can have on its caller's environment. We have implemented this approach by modifying the mechanics of tag references (module calls) in PushGP, and we have incorporated these changes in GLEAM. In this approach, modules are executed with the stack environments that are isolated from the stack environments of their callers. Modules can modify stacks in arbitrary ways as they execute, but when a module code completes execution, the stacks revert to the values they had at the time of the module call, except for a single value that is pushed onto or popped off of a single stack. The single change that will persist is the last stack operation performed before the end of the module's execution.

The modules in many genetic programming behave like functions in the general-purpose programming languages: the execution of the module does not affect the variables (for example, stacks, registers, etc.) in the rest of the program. The module returns a single output value which is used by the calling program. In PushGP, however, a module is simply a sequence of instructions grouped together and labeled with an integer tag. Whenever the module is executed, it can change multiple stacks, and all of those changes persist after the execution of the module is complete. The concept of environments is meant to modify the structure of modules in PushGP so that they also behave like modules in other genetic programming systems.

Environments allow code to manipulate the state of the system, often represented by different variables (or stacks, as in the case with PushGP), in a limited scope. In general, environments take arguments and produce return values like functions; however, these behaviors are limited in this work to make them more useful to modules, as described below. We will explain the environments in detail using their implementation in PushGP.

5.3.1 Environments for Modules

The state in Push programs at a particular point during execution is described by the data present on the stacks. These stacks can be modified in two ways: one or more values can be pushed onto the stacks, and one or more values may be taken off the stacks (usually from the top of the stacks). The Push state reflects both the data stored at any moment during execution as well as the extent to which the execution has progressed since the program (or the part of the program that has not been executed yet) itself is stored on the `exec` stack as it executes.

When a module is called, it takes as input the current state of the stacks, changes the contents on the stacks, and produces a modified stack state. Consequently, with a single module call, which may execute multiple instructions at once, multiple stacks may be changed in significant ways. We would like modules to encapsulate single behaviors in the same way that other Push instructions do; instructions typically have more limited effects on stacks, perhaps popping arguments from one or two stacks and pushing a return value onto one stack. Moreover, functions or methods in most programming languages, when executed, produce single output values instead of changing larger parts of the state. In order to limit the effects of modules and, therefore, their scope to produce changes in state, we introduce the use of environments.

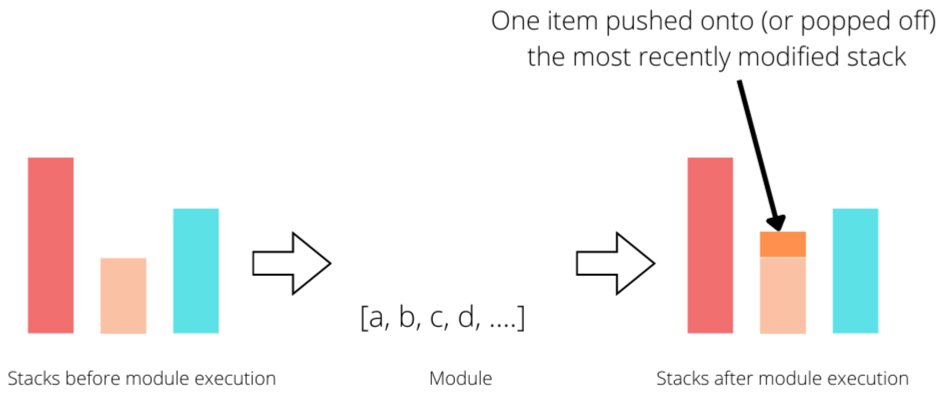


Figure 5.4: Depiction of stacks in PushGP before and after module execution with environments.

Environments in PushGP work in the following way. Whenever a module is called by the program, before the code in the module is executed, the state of the program is saved (call it `oldState`). Then the code of the module is executed in the original state, during which the stacks are modified. We keep track of which stacks are modified and in which order. Once the execution is complete, we look for the last stack that was modified (call it `lastModifiedStack`) and whether a new value was pushed onto that stack or removed from it. Now, before exiting the module, we revert back to the original state of the program (`oldState`) with only the `lastModifiedStack` stack changed with either one new value being pushed or one existing value popped. In Figure 5.4, we depict the state of a Push program, represented by stacks of different types, before and after the execution of a module.

Let us take a code segment written in Push as an example to illustrate how environments work: `(1.0 "bravo" float_sin string_reverse)`. Without environments, the output of this code segment would be 0.84 on the `float` stack and “ovarb” on the `string` stack. If the same code segment is executed with environments, the output would just be “ovarb” pushed onto the `string` stack since that was the stack most recently modified. Since the stacks were empty before the code segment started executing, they will be empty afterward as well, except for one change to the `string` stack, i.e., the item "ovarb" pushed onto the `string` stack.

Please note that even the execution of a single instruction with and without the environments is different. An instruction in Push usually pops off the arguments from the stacks before applying computation on them and then pushes the result on some stack. Instructions with environments, on the other hand, will not typically pop the arguments off the stacks, since the last change of pushing a return value will be the only difference that the environment makes on the original state.

5.3.2 Environments for Programs

The concept of environments need not be limited to how it is used within modules. Environments could be used independently within the programs as well to allow programs to produce locally-scoped changes. For example, a program could use an environment to execute some otherwise destructive code, allowing it to produce a value that could be useful in the remainder of the program. We have implemented Push instructions that can signal the beginning and end of environments. While experimenting with such 'inline' environments is beyond the scope of this chapter, their use could be explored independently of GLEAM in future work.

5.4 GLEAM as a platform for testing

Due to its flexible architecture and tunable parameters, the GLEAM framework can be used as a general-purpose platform to test various aspects relating to modularity in genetic programming: how to use already-generated modules, how to generate new ones, how to decide which old modules to replace, where do the modules actually reside, etc. We put these methods into four categories and discuss some examples of each of them. The categories are listed in Table 5.1 and are described below:

Table 5.1: Some of the methods that can be tested using GLEAM framework.

Using modules	Storing modules
Reference by an identifier Direct insertion	In the program A library local to the program A library shared by all individuals
Generating modules	**Replacing modules**
From the program using the modules From other programs External repository	Replace the ones not used by the program Oldest

1. **Using modules:** How do modules get referenced? The methods to do that may include calling the modules using their unique identifiers, inserting them directly into the program, etc.

2. **Storing modules:** Where do modules reside? They might be kept in a library local to an individual, a library common to all the individuals in the population, or a combination of both. They can also be defined as part of the program itself.

3. **Generating modules:** Where do new modules come from? They can come from the segments of the program using them or any other program in the population or any external repository. They can even be generated randomly from the instruction set available to the program.

4. **Replacing modules:** Which modules get replaced whenever new modules are being added? Various policies that can be adopted are: replace the ones not used by the program, replace the ones least recently used, replace the oldest ones, etc.

5.5 Experiments and Analysis of GLEAM

In this section, we describe some experiments conducted to demonstrate the effectiveness of GLEAM in evolving modular programs and to test various methods relating to the generation, usage, etc. of modules during evolution.

5.5.1 Experimental set-up

All experiments were conducted in a version of PushGP written in Clojure, called Clojush[1]. To test the effectiveness of GLEAM, we ran it on all the problems from the General Program Synthesis Benchmark Suite [23], also known as PSB1. These problems have varying levels of difficulty based on the success rate of the state of the art [23] on them. They also use multiple data types in inputs, outputs, and evolved solutions. Details of the problems are given in [23]. But, to give a sense of the difficulty level of these problems, we provide descriptions of some of them, chosen randomly from the suite, here:

- Last Index of Zero: Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.

- Count Odds: Given a vector of integers, return the number of integers that are odd, without the use of a specific even or odd instruction (but allowing instructions such as mod and quotient).

- Compare String Lengths: Given three strings $n1$, $n2$, and $n3$, return true if $length(n1) < length(n2) < length(n3)$, and false otherwise.

- Small or Large: Given an integer n, print "small" if $n < 1000$ and "large" if $n \geq 2000$ (and nothing if $1000 \leq n < 2000$).

[1]https://github.com/lspector/Clojush

Table 5.2: Genetic Programming Parameters.

Parameter	Value
Population size	1000
Number of generations	300
Parent selection algorithm	lexicase
Mutation operator	UMAD
Mutation rate	0.09
Genome Representation	Plushy
Number of runs per condition	100

- Double Letters: Given a string, print the string, doubling every letter character, and tripling every exclamation point. All other non-alphabetic and non-exclamation characters should be printed a single time each.

Some of the genetic programming parameters used for the experiments are given in Table 7.7. For other parameters, we use the default values given in the problem description files in Clojush github repository[2] For the problems in the benchmark suite, the best results [18] so far have been obtained using lexicase selection as selection operator [24], uniform mutation by addition and deletion (UMAD) [21] as the mutation operator, and no crossover operator. We use the same selection method, but in addition to UMAD, we also apply Uniform Deletion with a rate of 0.01 to the programs only when using GLEAM. This is done to offset the increase in the size of the programs due to the absorption of modules. We did some preliminary experiments with different rates for uniform deletion and found that with the GLEAM settings we use, the rate of 0.01 worked the best. In other words, this rate prevented the program sizes from converging to zero or the maximum size.

[2]The only change is the "evalpush-limit" for Sum of Squares problem, for which we use 2000 instead of the default value of 4000.

5.5.2 Using GLEAM to evolve modular programs

As far as parameters specific to GLEAM are concerned, we conducted some preliminary experiments with various settings, and the configuration that worked the best, in terms of the number of successes on the problems from the benchmark suite, is given in Table 5.3. In this configuration, the mutation operator used for the modules is the same as that of the program, i.e., UMAD, but the rate used for the modules is half of what is used for the program. Initial modules have a size one-tenth the size of the initial programs. Every unused module is replaced with a probability of 0.75 by a new module. When replacing an unused module, the new modules come from two places:

1. 75% of the time, a random sequence of maximum length 10 is taken from the program genome. We make sure that the parentheses are balanced, and if not, we only take the sub-sequence with the balanced parentheses.

2. 25% of the time, a random individual is chosen from the population, and the module which has been used continuously by the program for the longest time (measured by the number of generations) is chosen.

The extraction of code segments from the program is carried out in the following way. First, a number is chosen randomly between 1 and 10. This number called, $len\_segment$, serves as the tentative length of the new module. Then, a random point in the program genome is chosen. Let's call it $start$. Now, the segment between $start$ and $min(start + len\_segment, len\_program)$ is extracted. After extracting the module, we check whether it has balanced parentheses. If not, we choose the first subsegment of the extracted segment with balanced parentheses as the new module. Note that the new module might also be empty and, in that case, we do not replace the old module.

Table 5.3: GLEAM Parameters.

Parameter	Value
Number of modules per individual	10
Mutation operator for modules	UMAD
Mutation rate for modules	half of what is used for the program
Extraction method	sequences from the same program & modules from other programs
Extraction rate	0.75
Module Absorption rate	0.1

We also allow module references to get absorbed in the program as well as in other modules with a probability of 0.1. That means one out of every ten module references, one would be expanded.

Table 5.4 gives the number of successes, out of 100, for various problems. The second column gives the number of successes without using GLEAM. The third column corresponds to the parameter settings of GLEAM given in Table 5.3. The last column shows the improvement in the success rate of various problems due to GLEAM, and the problems have been ordered according to this column. To qualify as a solution, the program must have a zero error on the training set, which was used during evolution, and also on a held-out test set, which was not used during evolution. Specifically, the programs which produced zero error on the training set were first simplified according to the procedure detailed in [20], and then were run again on the test set. Those simplified evolved programs producing zero error on the test set were termed solutions. Automatic simplification was done because it has been shown to improve generalization on the unseen test set.

To check whether the improvement in the success rates is statistically significant, we use Wilcoxon Signed Rank Test. It is a non-parametric test that does not assume that the differences between paired samples are normally distributed. It takes into account the sign (increase or decrease) as well as the amount of improvements

Table 5.4: Number of successes out of 100 without and with GLEAM. The last column shows the improvement in the success rate due to GLEAM.

Problem	W/out GLEAM	With GLEAM	Improvement
Compare String Lengths	29	47	18
Median	58	71	13
Double Letters	34	46	12
X-word Lines	60	71	11
Scrabble Score	12	19	7
Vector Average	86	93	7
Small or Large	5	9	4
Checksum	0	3	3
For Loop Index	0	2	2
Mirror Image	98	100	2
Vectors Summed	16	17	1
Grade	0	1	1
Smallest	99	100	1
Last Index of Zero	65	65	0
Count Odds	11	11	0
Even Squares	0	0	0
Pig Latin	0	0	0
Digits	11	10	-1
Number IO	100	99	-1
String Lengths Backwards	91	90	-1
Super Anagrams	4	1	-3
Replace Space with Newline	93	89	-4
Sum of squares	24	17	-7

across multiple problems. This test has been recommended for statistically comparing different classifiers on multiple datasets as well [11]. Applying this test on the improvements due to GLEAM, we find that they are statistically significant at 0.05 level.

In order to better visualize the difference in the success rates with and without using GLEAM, we plot the experimental results in Figure 5.5 as well. We also augment the bars corresponding to various problems under different conditions with confidence intervals. We use binomial proportion confidence intervals estimated using the Wilson score method [60].

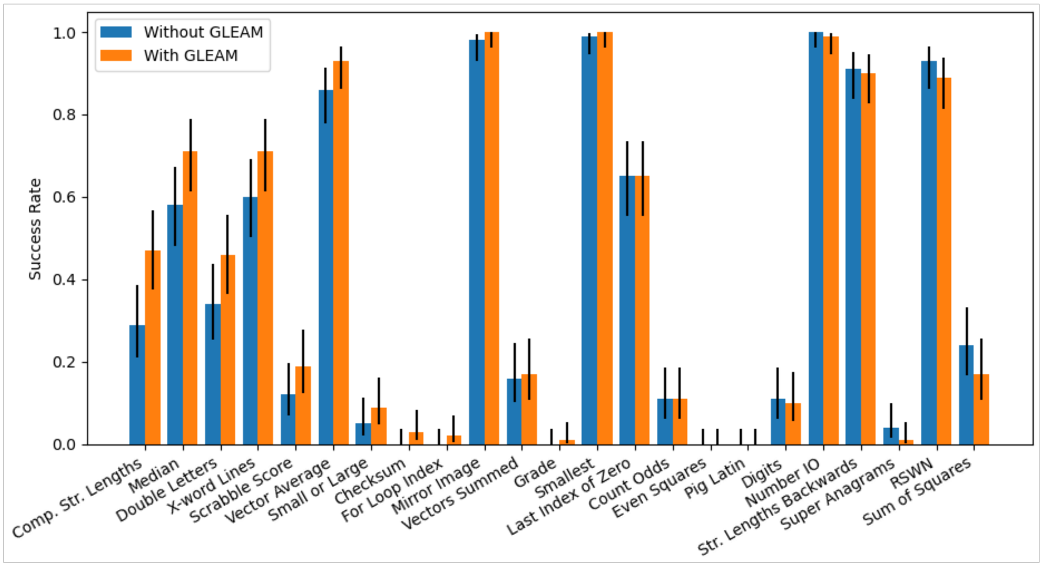


Figure 5.5: Success rate (number of successful runs divided by the total number of runs) for various problems with and without using GLEAM. The errors bars correspond to the Wilson Score confidence intervals.

Note that for three problems, namely, Checksum, For Loop Index, and Grade, the system managed to get a solution with GLEAM when there was none without using GLEAM. This is quite significant as GLEAM 'discovered' the solutions to these problems. The significance of these results is not clear from the significance tests we have done.

The future work will entail testing various methods available in the literature for generating new modules, storing those modules, etc., to further improve the success rates.

5.5.3 Analysis of modules evolved using GLEAM

In this section, we will analyze different aspects of the modules evolved using GLEAM on the problems mentioned in Table 5.4. To that end, we ask the following questions:

Do modules contain useful information? In other words, we check whether the instructions contained in the final evolved modules for solutions are *useful* for the given problem. We measure *usefulness* in a simple way. The instructions which are present in the simplified solution programs are considered useful. This is because simplification removes unnecessary instructions from the program leaving behind instructions that are absolutely required for the solution programs to solve the problem at hand. Therefore, we look at the percentage of instructions in the modules that are also present in the simplified solution programs. The formulation that we use to calculate it is given below:

$$usefulness = \frac{(instructions\ in\ modules) \cap (instructions\ in\ simpl.\ solutions)}{\#(instructions\ in\ modules)} \tag{5.1}$$

We report the data in Table 5.5. For most of the problems, the usefulness score of modules is quite high. This indicates that the instructions in modules are not arbitrary and are indeed related to the problem being solved.

Do programs use modules? We answer this question in two ways. First, for each solution program, we calculate the number of module references that have been absorbed in the programs at various points in its lineage. We average this number for all the solutions. Second, we also look at the number of simplified solution programs that have at least one module reference. We report these numbers in Table 5.5.

Since the solution programs and the individuals in their lineages do use modules according to Table 5.5, it is likely that these modules contain some useful information that was used by them. It might also be the case that these modules might have helped the individuals find new pathways to solutions through the search space. Further research is needed to verify these claims.

Table 5.5: Various attributes of solutions related to modules for different problems.

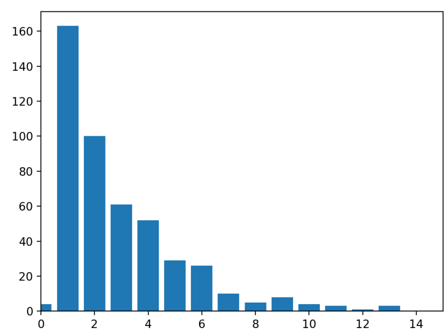
Problem	%age of useful instructions	#modules absorbed	%age of sols. with module calls
Compare String Lengths	78.50	6.74	17.02
Median	55.91	2.66	32.39
Double Letters	59.59	15.54	54.35
X-word Lines	79.43	12.25	14.08
Scrabble Score	26.71	14.47	5.00
Vector Average	89.36	6.01	17.20
Small or Large	80.43	5.44	33.33
Checksum	18.52	14.00	0.00
For Loop Index	30.00	8.50	50.00
Mirror Image	17.43	1.03	17.43
Vectors Summed	50.72	17.00	17.65
Grade	42.31	11.00	0.00
Smallest	30.19	0.70	20.00
Last Index of Zero	60.16	2.34	7.69
Count Odds	26.09	5.45	18.18
Even Squares	NA	NA	NA
Pig Latin	NA	NA	NA
Digits	47.92	4.70	10.00
Number IO	14.87	1.07	18.18
String Lengths Backwards	83.23	3.33	10.00
Super Anagrams	25.00	20.00	0.00
Replace Space with Newline	50.77	6.42	7.86
Sum of squares	36.84	9.47	5.88

Although the numbers presented in Table 5.5 are indicative of the fact that modules contain some useful information, these numbers do not correlate highly with the actual number of successes for a given problem. Further study will be required to understand the significance of these results, which may depend on specific aspects of problems that affect the utility of modules.

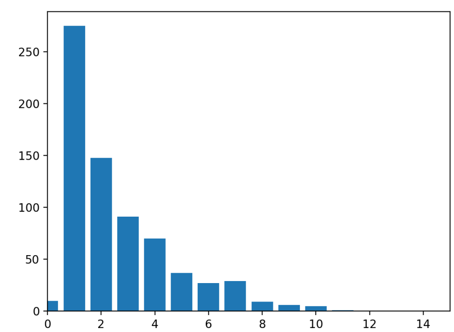
What is the size distribution of modules? We also look at the sizes of modules evolved using GLEAM. The size of a module, or any code segment in general, in Push is defined as the sum of the number of instructions and parentheses pairs present in it. Since we have capped the number of instructions in a code segment taken from the program during extraction at 10, the size of the modules would usually be less than 10 unless there has been addition of more instructions during mutation.

Note that only the modules present in the libraries of the final solutions are used for the analysis; the modules used by the programs in the lineages of these solutions are not considered. For the analysis, we choose the top two and bottom two problems from Table 5.4 to include a diverse set: the problems for which GLEAM improved the success rate considerably and the problems for which GLEAM did not do so. The plots are shown in Figure 5.6.

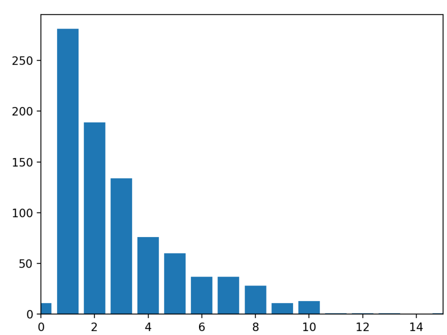
For the problems studied here, the distribution is somewhat exponential, with modules of size 1 appearing the most. Incidentally, the problems not shown here also have similar distributions. This means that with the current setting of GLEAM, evolution favors small modules over big ones. Further research is needed to figure out the reasons behind this phenomenon. Modules of size 0 can occur when the mutation operator removes all the instructions from a particular module. Although these modules do not contain any instructions, they behave as regular modules in the sense that the mutation operator might add new instructions to them or they might get replaced by new modules.



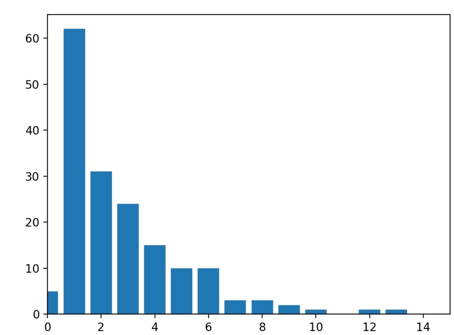
(a) Compare String Lengths



(b) Median



(c) Replace Space with Newline



(d) Sum of Squares

Figure 5.6: Size distribution of the evolved modules for various problems. The values on the x-axis have been clipped at 15, and the y-axis shows the number of modules of different lengths in the final solutions.

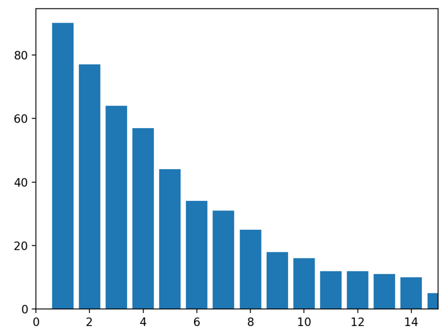
How long do programs use the modules? There are two ways in which references to modules can be removed in a given program: mutation might remove the reference by deleting it, and the module might get absorbed. Here, we look at the number of generations modules are used continuously by individuals (in a particular lineage) before being removed by mutation or absorption. We plot the number of generations in Figure 5.7 for solution lineages. The x-axis shows the number of generations, and the y-axis shows the number of modules that were used continuously for given generations before being removed.

The plots show that most of the modules are not used by individuals for a long time; new references are constantly being inserted, while old ones are either removed through mutation or absorbed. Note that the numbers shown in the figure are highly dependent on the mutation and absorption rates, and therefore, we may not derive useful conclusions from these plots. The numbers, however, can still be useful in verifying that GLEAM is working as expected and which rates, if any, need to be changed.

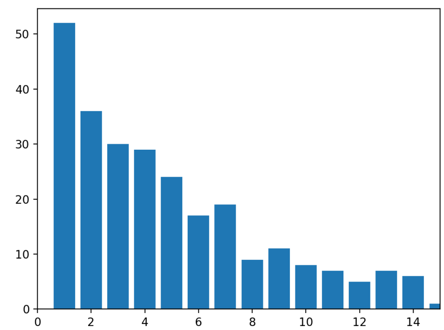
5.5.4 Using GLEAM as a testing platform

We also conducted some additional experiments to illustrate the effectiveness of GLEAM to be used as a testbed for testing various methods of generating, propagating, and using modules. To that end, we experiment with some of the conditions described in Section 5.4. We call the parameter settings of Table 5.3 "Config A" and compare the performance of all other settings with this one. Note that we use only 50 runs for the experiments in Table 5.6, and these runs are independent of the runs in Table 5.4. In each of the configurations described below, only one parameter is changed while keeping everything else the same as in Config A.

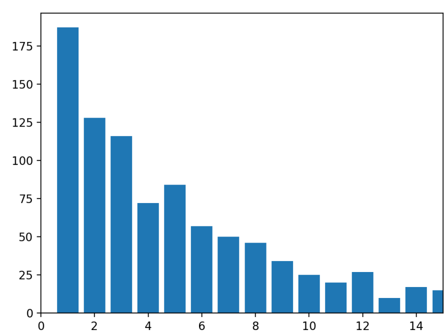
1. **Config A**: The parameter setting of Table 5.3.



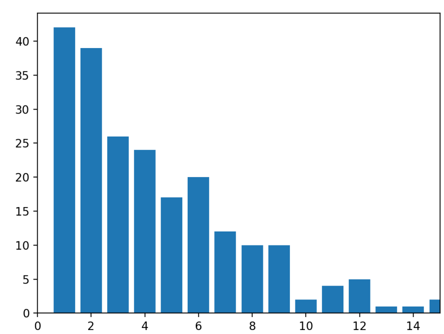
(a) Compare String Lengths



(b) Median



(c) Replace Space with Newline



(d) Sum of Squares

Figure 5.7: Distribution of the number of generations modules were used continuously by individuals in solution lineages. The x-axis shows the number of generations, and the y-axis shows the number of modules that were used continuously for given generations. The values on the x-axis have been clipped at 15.

2. **Config B (Random Segments)**: In this configuration, during extraction, 25% of the time, new modules still come from a randomly selected individual, as discussed in Section 5.5.2, but the other 75% of the time, randomly generated code segments become new modules. Code segments from the same program are not used as new modules. To keep the comparison fair, we calculate the lengths of the new modules as if they are coming from the program according to the procedure described in Section 5.5.2. After that, the genes are generated randomly instead of getting extracted from the program.

3. **Config C (No Absorption)**: We set the absorption rate to be zero. This means there is no absorption either in the program or other modules.

4. **Config D (High Absorption)**: We set the absorption rate to 0.25. This is meant to test whether higher absorption is better or not.

5. **Config E (Modules are not mutated)**: There is no mutation at the level of modules.

The results are given in Table 5.6. Since the changes in the success rate are moderate and may not be statistically significant, we will mainly talk about some general trends rather than any concrete conclusions drawn from the data. The following trends may be observed:

1. Using segments extracted from the program itself as modules seems to be better than using randomly generated segments.

2. Some absorption is better than having no absorption or very high absorption.

3. Having mutation at the level of modules seems to work better than not mutating modules at all.

Table 5.6: Number of success out of 50 for various configurations.

Problem	Baseline	Config A	Config B	Config C	Config D	Config E
LIZ	33	34	24	36	38	35
Count Odds	3	10	5	7	5	5
CSL	14	23	17	18	20	11
Double Letters	13	19	19	17	22	11
Small or Large	3	5	4	7	4	4

5.6 GLEAM in GE

So far, we have described the implementation of GLEAM in PushGP, but it can also be implemented in other similar systems. In this section, we discuss its implementation in a program synthesis system called Grammatical Evolution (GE) [40].

GE is a type of genetic programming technique where genomes in the form of integer strings are mapped to programs using a given grammar in Backus–Naur form (BNF). At every step of the translation phase, the integer value at each location is used to determine which rule from the grammar should be used to expand a non-terminal in the partially built program.

In this section, we present one of the ways GLEAM can be implemented in GE. We first modify the grammar in the following way. For each non-terminal that contains at least one non-terminal in its production rule, we add an additional production that contains a non-terminal specifying a set of modules. We also had the corresponding production rule that expands the non-terminal into a set of modules. See Figure 5.8 for an example.

The right-hand side of the production rules corresponding to the modules (e.g., <acts_lib> in Figure 5.8) is different for different individuals. The collection of these production rules serves as the local library for the individual. The location of a module in the production rule serves as its label or identifier.

```
<acts> ::= <act>| <act><acts>
<act>  ::= move|left|right
```

(a) Original Grammar

```
<acts>     ::= <act>| <act><acts> | <acts_lib>
<act>      ::= move | left | right
<acts_lib> ::= move move | left move right
```

(b) Modified Grammar

Figure 5.8: (a) Original Grammar that is used in the system without GLEAM. (b) Modified Grammar that is used by individuals in the systems with GLEAM. The last production rule in (b) is different for different individuals. Both grammars are adapted from [58].

Although our procedure to modify the grammar is similar to what is given in [58], there is one significant difference. In our system, the production rules corresponding to the modules are different for different individuals, whereas, in most other GE systems employing modularity, the same grammar is used by all the individuals in the population. In other words, in our system, modules are local to a given individual, whereas in other systems, modules are shared by all the individuals in the population.

The **extraction** operation is carried out in the following way. For every module that needs to be replaced, we look for the corresponding non-terminal in the derivation tree of the program. If the non-terminal is present in the derivation tree, we take the output on the subtree situated at that non-terminal node and use it as a new module. If we do find the non-terminal we are looking for, we do not replace the corresponding module. In the method that we specify here, there is currently no way to do **absorption** and **mutation** of modules, but that can be taken up as future work.

5.7 Multi-task GLEAM

In this section, we propose an extension of GLEAM called multi-task GLEAM. This framework is used to solve multiple problems simultaneously with GLEAM.

The motivation behind this idea is that there might be some modules that are useful for many related problems and, therefore, can be put in a global library shared by different problems. Take, for example, the modules in standard libraries for conventional programming languages, which are used by many related problems. Moreover, the utility of a code segment across problems might make it a good candidate for a module to be added to the global library.

Multi-task GLEAM can be implemented in the following way. The evolving population can be partitioned into different parts, where each part attempts to solve a different but related problem. For example, if we are trying to solve two problems, half of the population is devoted to solving one problem, whereas the other half attempts to solve the other problem. We keep a Global Library of modules which is used as a source of new modules for the individuals in the population during extraction. Basically, during extraction, in addition to using the code segments from the programs as new modules, the modules from the Global Library can be imported into the local library of an individual.

The global library is initialized and updated in such a way that it contains modules that are 'useful' to all the problems the population is trying to solve. For example, if we are trying to solve multiple problems that work with vectors of integers (Count odds, Last index of zero, etc. from the benchmark suite), the code segment that calculates the length of the vector, or some other instructions that can iterate over a given vector, etc. are good candidates for insertion into the global library.

Given a population of individuals (programs as well as the corresponding local libraries) solving multiple problems, how to extract the modules to be inserted into the global library? We describe some of the methods to do that here:

1. *Frequent sequence-based*: These methods basically search for the most frequently occurring code segments in the programs. One of the main issues with this approach is that some languages, including Push, are somewhat order-agnostic, meaning multiple permutations of the programs might be functionally equivalent. Therefore, these methods are prone to underestimate the frequency of certain code segments.

2. *Frequent itemset-based*: These methods consider everything as sets. So basically, the programs would be bags of instructions. This is also problematic because it completely disregards the order of instructions.

3. *Summarization-based*: If we consider Push programs as text, we can use summarization methods to extract a 'summary'.

4. *Keyword detection-based*: We can use a type of summarization where we extract the most useful/representative keywords or phrases from a text. It has the same issues as with the methods based on frequent-sequences extraction.

5.8 Cultural Tags

We implemented and tested many other frameworks which are similar to GLEAM in principle but did not produce significant improvements in the performance of genetic programming systems. We discuss one such idea here, called Cultural Tags.

Cultural Tagspace is basically a global library of labeled modules that can be used by any individual in the population, but only certain individuals which satisfy some conditions are allowed to 'update' it, i.e., add new modules or replace old ones. It differs from GLEAM in one important aspect: whereas in GLEAM, every individual has its own local library, in Cultural Tagspace, there is a global library shared by all the individuals in the population.

'Culture' as a concept used in a subfield of biological evolution called cultural evolutionary theory essentially means transmitting information from one generation to the other through non-genetic means. In other words, culture may be defined as collection of ideas, behaviors, and artifacts which can be transferred between individuals by the processes of teaching and learning [4, 10]. The main feature of 'non-genetic transmission' is that an individual can use a certain artifact even when it did not come from its parents.

In genetic programming, culture can mean the discovery of useful code segments that can be used by individuals in the later generations. It has been implemented as a shared indexed memory that can be used and modified by all individuals during the process of evolution [55].

In this section, we describe the implementation of culture using tags that were introduced in [56, 53]. Tags can be defined as identifiers used by an evolving program to store code blocks and reference them later on. They provide a mechanism for the evolving programs to share their code with other programs across generations.

5.8.1 Cultural Inheritance in Genetic Programming

In its most basic implementation, i.e., a shared indexed memory [55], culture has been shown to reduce the amount of computational effort required to solve some simple problems like symbolic regression of $y = x^4 + x^3 + x^2 + x$ and action selection in Wumpus World. Other similar ideas in genetic programming, though they do not use the concept of culture explicitly, include the use of run-transferable libraries [25], in which useful functions discovered in one run are available for use in the next run.

Culture can prove beneficial in the evolution of programs as it provides a mechanism to pass information across generations by non-genetic means. It is not subjected to genetic operators like mutation and hence can be used to pass from one generation to the other the knowledge which would otherwise be difficult or impossible to be transmitted through genetic means.

Implementing culture in a given genetic programming system requires the presence of the following mechanisms:

- A mechanism in the form of a data structure to store modules and index them by some identifier. Examples of such data structures include dictionaries in Python, hashmaps in Clojure, etc.
- A mechanism to make accessible the contents of the culture to all individuals across all generations.
- A mechanism to update the culture at regular intervals.

5.8.2 Cultural Tagspace

We propose the idea of cultural tagspace, which may be defined as a collection of code segments indexed by a number (called its 'tag'). Any individual belonging to any generation can retrieve a particular code block using its tag.

We implement tagspaces as associative arrays (also known as maps or dictionaries in different languages), where tags are keys and code segments are the values. Tags are integers between zero and a pre-set maximum. Programs can use three kinds of instructions to interact with a tagspace: to store a code segment with a given tag, to retrieve a code segment with a particular tag, and to remove an association. While accessing an association, if there is an entry with the given tag present in the tagspace, the corresponding code segment is returned. Otherwise, we use a type of inexact matching given in [56]: basically, we find a tag greater than but closest to the given tag and return the corresponding code segment; if we reach the maximum tag number, we start from zero.

Our implementation of cultural tagspace lets all individuals across all generations use the tagspace, but only 'qualified' individuals are allowed to update it. An individual being qualified means it has to meet certain conditions for it to be able to change the cultural tagspace. This is done to prevent the possibility of edits by individuals which have very low fitness and may even contain harmful code.

The cultural tagspace gets updated in the following way. The tagspace is initialized once for every run. All individuals in a generation are executed one after another. Each individual creates a 'local' tagspace with the content copied from the cultural tagspace. The individual is then executed on given test cases one by one. The local tagspace modified during execution on one test case is available for execution with the next test case. After executing the individual on the last test case, the modified tagspace is allowed to update the cultural tagspace only if the individual is 'qualified.' The updated tagspace is then used for the execution of the next individual, and the same process is repeated for all the individuals in the population.

5.8.3 Qualifications to update the Cultural Tagspace

Some of the possible qualification criteria for individuals to update the cultural tagspace include:

1. All individuals are qualified, i.e., the changes made by an individual in the local tagspace are transferred to the cultural tagspace unconditionally.

2. An individual is qualified if it is better than all its parents on at least one test case; it can be worse on other test cases.

3. An individual is qualified if it is not worse than its parents, i.e., on every test case, its error is less than or equal to the error of all its parents.

4. An individual is qualified if its performance is affected by the absence of cultural tagspace, or, in other words, if its error vector is different with and without using cultural tagspace. This way, only the individuals which actually make use of cultural tagspace are allowed to update it.

If the individual is not qualified, its local tagspace has no effect on the cultural tagspace. Note that the changes made by individuals in the first generation are directly adopted by the cultural tagspace since they do not have parents.

5.8.4 Different ways of updating Cultural tagspace

Cultural tagspace can be updated in one of two ways: local tagspace becomes the new cultural tagspace and the old cultural tagspace gets deleted, or the local tagspace gets merged into the old cultural tagspace. The process of merging can proceed as follows. If there is a tag in the local tagspace which is not present in the cultural tagspace, it is directly copied into the cultural tagspace. Otherwise, the code segment in the local tagspace replaces the corresponding code segment in the old cultural tagspace if it satisfies certain conditions. One of the possible conditions can be that the code segment belongs to an individual with better errors than the individual which produced the code segment it is trying to replace.

5.9 Conclusions

We presented and analyzed a general framework for evolving modular programs in genetic programming called GLEAM. Although it can be applied to any genetic programming system, in this paper, we study its effects in PushGP, a genetic programming system that evolves programs in Push programming language. We find that GLEAM significantly improves the success rate on multiple benchmark problems. We also describe how GLEAM can be used as a platform to test various methods of generating, using, storing, and replacing modules during the evolution of modular programs. We test some of these methods experimentally as well. We also show that solutions evolved using GLEAM often use modules, indicating their usefulness during the process of evolution. Some possible extensions of GLEAM—in a related genetic programming system called Grammatical Evolution, and in a setting where multiple problems are solved simultaneously—are also discussed.

CHAPTER 6

WHY ARE MODULES USEFUL?

As described in Chapter 2, there are many genetic programming systems that evolve modular programs. The ability to evolve modular code has also improved the performance of some of these systems. GLEAM, as described in Chapter 5, has led to significant improvements in the success rate on a number of benchmark software synthesis problems, which often use complex programming constructs like loops and use multiple data types.

Although these systems, including GLEAM, have managed to evolve modular programs, the reasons for why modules are useful, at least in the systems in which they lead to improvement in the performance of the systems, have not been explored in detail. In this chapter, we present a set of hypotheses for why modules might be useful in genetic programming systems in general and GLEAM architecture as implemented in PushGP in particular.

6.1 Experimental Set-up

We use the GLEAM architecture to test the hypotheses in PushGP. As mentioned elsewhere, the performance of a genetic programming system for a particular problem is usually measured in terms of success rate, which is defined as the number of successful runs out of total runs launched for that problem. Successful runs are the ones where the system has evolved a program having zero errors on all test cases. Currently, the threshold is zero, but it can be changed according to the level of difficulty of the problem.

We take the best configuration of GLEAM found so far in Chapter 5 and call it the 'standard' configuration. We will make changes to this configuration based on the hypothesis getting tested and observe the change in performance of the system.

Since we will be running multiple experiments under different parameter settings, to keep the runtime of experiments reasonable, we choose eight problems from the PSB1 benchmark suite [18] instead of all the problems in that suite. We choose these problems based on the following criteria:

- They have a medium difficulty level (in terms of the success rate of state of the art on them).
- They employ multiple data types in solutions.
- It usually takes less than a day for a typical GP run launched for them to finish.
- GLEAM has either improved the success rate on these problems, or kept it the same as before.

6.2 Why are modules useful?

In regular programming languages, functions are used to provide **reusability** and **abstraction** [13, 27]. Whereas reusability refers to the ability to use the same code again and again without having to write that code multiple times in the program, the abstraction allows the users to use a function with just knowing *what* it does; they do not need to know *how* it does it.

John Koza, in his first book on Genetic Programming [30], explains the advantages of defining and using functions in terms of not having to "write numerous lines of essentially similar code" and enhancing "the understandability of a program because it highlights common calculations." Additionally, functions may also help to decompose the problem into smaller, simpler sub-problems.

Based on the previous work on modularity described in Chapter 2 and elsewhere, in this section, we compile a set of hypotheses on why modules are useful in genetic programming and test them using GLEAM as implemented in PushGP.

6.2.1 Reuse code from the same program

Modules are useful because they give the evolving programs the ability to reuse a chunk of code multiple times without having to create that chunk again and again. This is similar to how function calls are used in human programming to execute a code chunk at multiple different places.

To test this hypothesis, we observe the change in performance of the genetic programming system when the evolving programs are restricted to using any given module only once. Basically, after applying all the genetic operators used in the standard case, we check whether there are multiple references to the same module. If there are any, we keep the first reference and delete all others from the program.

The results are given in Table 6.1 in the form of the number of successes for various problems under various conditions. If we apply the Wilcoxon Signed Rank test to the differences in the last two columns, we find that the differences are not statistically significant. Therefore, the reason behind GLEAM improving the success rate for the benchmark problems does not seem to be the ability it gives to the programs to call a module multiple times.

6.2.2 Reuse code from other individuals

Another aspect of the usefulness of modules that we explore is that modules allow evolving programs to use code segments from other individuals in the population. One of the ways of using code segments from other individuals in GLEAM is to use them during extraction. Basically, in GLEAM, whenever a new module is needed to replace an existing unused module, other than using code segments from the same

Table 6.1: Number of successes, out of 100, for various problems with GLEAM under two conditions: one where programs can have multiple references to a single module, the other where programs are restricted to use every module at most one time.

Problem	No GLEAM	GLEAM: Multiple references	GLEAM: Single reference
Last Index of Zero	65	65	67
Count Odds	11	11	23
Compare String Lengths	29	47	48
Double Letters	34	46	39
Small or Large	5	9	3
Median	58	71	57
Scrabble Score	12	19	19
X Word Lines	60	71	59

program, we can directly copy modules from a randomly chosen individual in the population.

Importing modules this way might be beneficial because other individuals in the population might have come up with 'good' modules which are useful to them as well as to the problem being solved in general. Directly copying such modules might be better than trying to build them from scratch. In software engineering as well, human programmers often use functions defined in other files, projects, software, etc., even if those codebases are not written by them or are not directly related to their current project.

To test this hypothesis, we make changes to the standard parameter settings such that modules from other individuals are not copied during extraction. Basically, in the standard condition, during extraction, 25% of the time, we select an individual from the population and copy the module which has been used by the individual for the highest number of generations. For the other 75% of the time, we select a random segment from the program as the new module. In the experimental condition, however, every time a new module is needed, we extract a random segment from the program.

Table 6.2: Number of successes, out of 100, for various problems with GLEAM under two conditions: one where programs can copy modules from other individuals during extraction, and the other where programs are restricted from doing so.

Problem	No GLEAM	GLEAM: Standard condition	GLEAM: No modules exchanged
Last Index of Zero	65	65	60
Count Odds	11	11	13
Compare String Lengths	29	47	29
Double Letters	34	46	42
Small or Large	5	9	2
Median	58	71	66
Scrabble Score	12	19	17
X Word Lines	60	71	70

The results are given in Table 6.2. Applying the Wilcoxon Signed Rank test on the last two columns, we observe that the success rate deteriorates significantly when programs are not allowed to use modules from other individuals in the population. This provides some evidence that being able to use code segments from other programs using modules is essential to achieving modularity with GLEAM.

6.2.3 Protection from frequent mutation

An evolving program might use a module to prevent a certain chunk of code from changing too frequently. Putting that code in a module will achieve this purpose as long as the genetic operators are applied on modules at a lower rate than the rest of the program.

This idea of changing modules with a lower rate of mutation is not new in the field of genetic programming. Take Hierarchical Genetic Programming [3], for example. It uses multiple levels of modules: modules that are called directly by programs are on Level 1, modules that are called by modules on Level 1 are on Level 2, and so on. Modules on Level 1 have a lower rate of mutation and crossover than the programs, modules on Level 2 have a lower rate of genetic operators than modules on Level 1, and so on. In fact, in many genetic programming systems, modules are not mutated at all, which effectively means applying the genetic operators with a rate of 0.

In software engineering, different functions are changed with different frequencies. The functions which are in-built into the language—for example, `len()` to calculate the length of a list in Python—are not changed at all. Other functions written by a human programmer in a given program are changed during development but not as frequently as the 'main' function.

This hypothesis has been tested by experimenting with different rates of genetic operators for modules relative to the programs that use them in GLEAM. In Table 6.3, we show the success rate for different rates of mutation for modules relative to the programs. In the table, the column with 'Ratio=0' implies that modules are not mutated, the column with 'Ratio=0.5' implies that the rate of mutation for modules is half of the rate for programs, and the column with 'Ratio=1.0' implies that the rate of mutation for modules is the same as that for programs. We compare the success rates in the last three columns with each other with the Wilcoxon Signed Rank test and observe that they are not significantly different from each other. This implies that providing 'protection' to the modules does not seem to be the reason behind GLEAM's success.

6.2.4 Hierarchical organization

Modules might also be useful to the evolving programs because they allow information to be organized in hierarchical or even more complex ways: programs can call modules, and modules can call each other as well as themselves. This type of complex control flow is not possible without modules.

Hierarchical organization of components might be necessary to build complex systems, in part because hierarchical systems can evolve faster than their non-hierarchic counterparts [49].

Table 6.3: Number of successes, out of 100, for various problems with GLEAM under three conditions: modules do not get mutated, modules get mutated with half the rate for programs, and modules get mutated with the same rate as that for programs.

Problem	No GLEAM	GLEAM		
		Ratio=0	*Ratio=0.5*	*Ratio=1.0*
Last Index of Zero	65	65	65	69
Count Odds	11	15	11	16
Compare String Lengths	29	33	47	37
Double Letters	34	45	46	42
Small or Large	5	10	9	8
Median	58	59	71	50
Scrabble Score	12	21	19	21
X Word Lines	60	65	71	55

We test this hypothesis by testing GLEAM under two conditions: one where modules can call each other, and the other where they cannot call each other. We implement the latter condition as follows. During initialization of modules and their subsequent mutation, the instruction set available to them does not include module referencing instructions. During extraction, we delete any module reference in the segment extracted from the program.

The results are given in Table 6.4. After applying the Wilcoxon Signed Rank Test, we find that there is no significant change in the success rates when modules are not allowed to call each other. This implies that the hierarchy of module calls is not one of the reasons behind GLEAM architecture's significantly better performance on benchmark problems.

6.3 Conclusions and Future Work

In this chapter, we presented a set of hypotheses on why modules might be useful to evolving programs. We tested them by changing different parameters in GLEAM and observing the change in performance as a result. From the experiments, some conclusions can be made: programs do not seem to use modules to reuse code segments,

Table 6.4: Number of successes, out of 100, for various problems with GLEAM under two conditions: one where programs can copy modules from other individuals during extraction, and the other where programs are restricted from doing so.

Problem	No GLEAM	GLEAM	
		With hierarchy	*W/out hierarchy*
Last Index of Zero	65	65	61
Count Odds	11	11	11
Compare String Lengths	29	47	41
Double Letters	34	46	47
Small or Large	5	9	13
Median	58	71	60
Scrabble Score	12	19	19
X Word Lines	60	71	73

programs use modules to import code segments from other programs, programs do not use modules to 'protect' code segments from frequent mutation, and programs do use modules to hierarchically store information.

Three main extensions of this work might be taken up in the future:

1. Since our experiments use the GLEAM architecture to evolve modular programs, the conclusions drawn from the results might be specific to GLEAM or the GP system that we are using, i.e., PushGP. To draw stronger conclusions, the hypotheses presented here can be tested in other modularity-inducing systems in the future.

2. Similar to the experiments performed in this chapter, we can test various hypotheses for why modules are difficult to evolve. Some of these hypotheses are mentioned in Chapter 4.

3. Since there are various costs associated with using modules, for simple problems, modules might not be that useful. In fact, even using instructions such as loops in genetic programming is not always useful in increasing the success rates for software synthesis problems [44]. But what exactly are the characteristics that

make certain problems benefit more from using modules than not using them at all? Some problems do seem to benefit from using modules, as demonstrated in the experiments in Chapter 5. Now, to answer the question raised above, we can look at the evolved solution programs for these problems and figure out the commonalities in them. The analysis can include data types used, various programming constructs employed to use those data types, etc., in these solution programs.

CHAPTER 7

CODE USE METRICS FOR EVOLVING PROGRAMS

The programs in genetic programming, just like in regular programming languages, can use a code segment repetitively, in the form of loops or some other similar structure. Here, we differentiate between two types of repetitive code occurrences in evolving programs: *code reuse*, where the code segment appears only once in the program but is executed again and again; and *code repetition*, where a particular code segment appears multiple times in the program at possibly different locations and these copies are executed separately.

In this chapter, we discuss two metrics—Reuse and Repetition—that measure the multiple occurrences of code segments. We describe the procedure for calculating them from program code and corresponding execution traces in PushGP. We discuss the utility of these metrics in two different settings. First, we use them as *design features*, which are basically structural features of programs that can be used alongside error measures to guide evolution. Our experiments demonstrate that using this technique leads to a decrease in the average size of successful programs. Second, we use the reuse metric to analyze the relationships between different parent selection methods, the numbers of individuals in the population that make use of loops, and success rates.

In the following sections, we introduce our metrics and the methods by which they can be calculated, particularly in PushGP. This is followed by a discussion on how the metrics can be used during evolution and what they can tell us about the relationship between parent selection methods and modularity. We finish with a summary of our conclusions.

7.1 Metrics to measure code use

In this chapter, we focus on the concept of modularity typically used in software engineering. Programmers often employ modules to solve parts of a given problem and reuse them multiple times to avoid code repetition. They also use many metrics to calculate the modularity of a given software. Coupling, for example, measures the interdependence between different modules; cohesion, on the other hand, measures the amount of interaction among components of a particular module [12]. A modular software design, therefore, is characterized by low coupling and high cohesion.

General-purpose programming languages like Python, Java, etc. contain constructs in the form of functions, classes, etc. that can be used to identify a module. Genetic programming systems, on the other hand, often do not have such standard constructs and generally use different ways of achieving modularity. Due to this, it becomes difficult to detect modules in different representations in genetic programming, and the metrics used in software engineering might not work for evolved programs.

Nevertheless, the metrics presented in this chapter are inspired by some of the concepts used in software engineering like code reusability, component-based development, etc. Reuse and Repetition—two metrics presented in this chapter—are also based on the heuristics humans use to write programs; in order to have less repetition, they try to reuse their code using features like functions, procedures, etc.

Since we want a formulation that can be used irrespective of the underlying genetic programming system, we use the information contained in the execution trace of a program to calculate the metrics. An execution trace is defined as the sequence of instructions executed at each step of the execution, and is often different from the program because of the presence of loops, conditionals, etc. Moreover, we maintain that instead of a single metric, there can be multiple metrics measuring different aspects of modularity, two of which are described in this chapter. Note that our formulation of the metrics assumes a program in the form of linear sequences of instructions, as opposed to trees or other similar data structures.

In this section, we first define the unit of code that we use during our calculations. And then, after going over the design principles used to formulate the metrics, we provide the exact equations to calculate those metrics.

7.1.1 Repeating Code Segment (RCS)

The source code can be decomposed into different units: tokens, structures, classes, functions, etc. The metrics discussed in this chapter use a unit called Repeating Code Segment (RCS). For the sake of simplicity, we will alternatively use the phrase *code segment* to describe the same concept. It is otherwise also known as an **unlabeled module** or simply a module ([41, 43, 42, 46]). For the purpose of this chapter, a single token, as well as a collection of tokens - keywords, variables, constants, etc. can be considered an RCS, provided the following conditions are met:

1. The order in which the tokens appear in an RCS should be the same as the order in which they appear in the program. For example, if we have a set of instructions `ABC` in the execution trace, for it to be considered an RCS, the same set of instructions should be present in the program in the same order.

2. The RCS should have a definite beginning and an end. For a group of instructions, as in most of the programming languages, including Push, this is indicated by brackets, keywords, or indentations. In the case of single instructions, however, no such construct is needed.

7.1.2 Design Principles for Code Use Metrics

In order to come up with the exact formulations for Reuse and Repetition, some design principles were used. The formulations which were in direct conflict with any of these principles were rejected, and those which complied with all of them were taken up for consideration. And the final set of metrics we present in Section 7.1.4 is one of them. The design principles are:

1. The frequency with which a repeating segment gets executed should contribute more than its size towards the metrics.

2. All consecutive sub-sequences of a repeating segment are also considered as repeating segments. For example, if `ABC` is an RCS, `A`, `B`, `C`, `AB`, and `BC` are also RCSs.

3. Since we are extracting segments from execution traces, there might be some instructions in the trace which were not there in the program. This can happen when an instruction from the program calls instructions from other programs or libraries. To keep things simple, we will not use such instructions in our calculation of the metrics.

4. The metrics measure the structural properties of a program. Therefore, the formulations of these metrics should not take into account the usefulness or other functional aspects of code segments.

5. The segments under consideration should appear or be used at least twice for them to be considered in the calculation.

6. Since both Reuse and Repetition consider the frequency and size of code segments, and they measure similar properties of those segments, they can have similar formulations. They should, however, be independent of each other. In

other words, a program can have a high value of Reuse with a low value of Repetition and vice-versa.

7.1.3 Reuse and Repetition

Reuse measures how frequently a copy of a code segment gets executed. Repetition, on the other hand, measures how frequently a given code segment appears in the program. The main difference between the two is that in the former one, one copy gets executed multiple times, whereas, in the latter, there are already multiple copies present in the program, each of which gets executed once.

A program will have high reuse if it has a function that is called repeatedly or a block of code that gets iterated multiple times. The program will have a high value of repetition if the same set of instructions is written multiple times and executed separately.

7.1.4 Reuse and Repetition from Execution Trace

An execution trace is a sequence of instructions arranged by the order in which they are executed. This sequence is often different from the actual program since the instructions at a particular position (say, a line number) in the program can call instructions at other locations.

The procedure to calculate the code use metrics from the execution traces is as follows.

1. Assign an identifier to every token in the program. For simplicity, let this identifier be the position of the token in the program.

2. Execute the program. Maintain two traces, one with instructions (called the *execution trace*) and the other with identifiers (called the *metadata trace*). During the course of execution, if we come across an instruction that was not present

in the program, we give it a special identifier. Such instructions will not be considered while computing the metrics.

Usually, in order to calculate the error vector of a given program, it is executed on a set of test cases. Consequently, we may have multiple execution traces for a single program. And since the metrics are calculated on the execution traces, we will have a list of values instead of a single number for each metric. To simplify this, we can use one of the two approaches: use summary statistics (for example, mean, maximum, etc.), or choose a value randomly from the list.

As mentioned above, we have two types of traces after the program gets executed. The exact procedure to extract repeating code segments from these traces depends on the language in which they are represented.

For the sake of simplicity, a repeating code segment is any group of instructions or identifiers that is repeated at least twice in the respective trace. Now, a simple and naive way to quantify the amount of reuse and repetition is to compute the proportion of execution trace under reuse or repetition. The corresponding formulation for Reuse (U) and Repetition (P) would be:

$$U = \frac{\sum_{i=1}^{m} l_i \cdot f_i}{l} \tag{7.1}$$

and

$$P = \frac{\sum_{i=1}^{n} l_i \cdot f_i}{l}, \tag{7.2}$$

where there are m segments being reused (i.e., there are m segments that are repeated at least twice in the metadata trace), n segments being repeated (i.e., there are n segments that are repeated at least twice in the execution trace), l_i and f_i are respectively the length and size of a given module i, and l is the length of the trace. The length of the trace is simply the number of instructions present in it, and the lengths

Table 7.1: Reuse values of some toy examples calculated using Equation 7.3 (in second column) and Equation 7.5 (in third column). Each letter denotes an instruction in the execution trace. We assume there is no repetition.

Execution Trace	Reuse (Eq. 7.3)	Reuse (Eq. 7.5)
`ABCABCDEF`	0.078	0.625
`ABCABCABCDEF`	0.019	1.250
`ABCABCABCABCDEF`	0.005	2.344
`ABCABCABCABCABCDEF`	0.001	5.000

of both the traces are equal. The main issue with this formulation is that it gives equal weight to the size and frequency of a module, which is in direct contradiction to the design principles given in Section 7.1.2.

After increasing the weight of the frequency and normalizing it accordingly, another possible formulation can be:

$$U = \frac{\sum_{i=1}^{m} l_i \cdot 2^{f_i}}{2^l} \tag{7.3}$$

and

$$P = \frac{\sum_{i=1}^{n} l_i \cdot 2^{f_i}}{2^l}, \tag{7.4}$$

where the symbols have the same meaning as given in the earlier formulation. This formulation is also problematic. Since the measures give more importance to the frequency of usage (exponentiation) than to the length of a module (simple multiplication), we get a higher value of Reuse when we have more frequent segments in an execution trace of a given length. In other words, as the length of the trace increases, the reuse measure starts to prefer short and frequent modules over bigger and less frequent ones. Table 7.1 illustrates this point. Although intuitively, reuse should increase since module `ABC` is getting reused more often, the actual reuse calculated from Equation 7.3 is decreasing. Similar issues occur during Repetition calculations.

The final formulations change the denominator to remedy this. The following equations are simplified versions of the equations given in [41]:

$$U = \frac{\sum_{i=1}^{m} l_i \cdot 2^{f_i}}{2^u} \tag{7.5}$$

and

$$P = \frac{\sum_{i=1}^{n} l_i \cdot 2^{f_i}}{2^v}, \tag{7.6}$$

where u is the number of unique identifiers used in the metadata trace, and v is the total number of instructions of the program used in the execution trace. In other words, in normalization, we use the number of unique identifiers or instructions used in the traces instead of the length of the traces. These numbers are also often different from the length of the program as some instructions get executed multiple times, and some might not get executed at all due to conditional operations. Again, while calculating u and v, we do not include the instructions that are not present in the program and appear only in the traces. See Table 7.1 for reuse values for some toy examples calculated using Equation 7.5.

7.1.5 Calculating Reuse and Repetition from Push Programs

As mentioned in Section 7.1.4, the procedure to retrieve repeating code segments from the execution trace of a given program depends on the language in which the program is represented. Since we are using PushGP in our experiments, we present the algorithm to extract the repeating segments from Push programs running on that system.

As mentioned in earlier chapters, Push is a stack-based programming language with separate stacks for every data type. During execution, the instructions can take their inputs from and place their outputs on different stacks. In each iteration, the top element of the execution stack gets executed. Hence, the sequence of the top elements

Table 7.2: A Push program example and the corresponding execution and metadata traces.

Program	`(exec_dup (exec_swap 1 2))`
Execution Trace	`((exec_dup (exec_swap 1 2)) exec_dup (exec_swap 1 2) exec_swap 2 1 (exec_swap 1 2) exec_swap 2 1)`
Metadata Trace	`((1 (2 3 4)) 1 (2 3 4) 2 4 3 (2 3 4) 2 4 3)`
Metadata Trace (modified)	`([1:4] [1:1] [2:4] [2:2] [4:4] [3:3] [2:4] [2:2] [4:4] [3:3])`

on the execution stack, also called the `exec` stack, after every iteration becomes the execution trace.

Algorithm 2, which describes the procedure to calculate the metrics for Push programs, can also be used for programs written in similar stack-based programming languages. Table 7.2 gives an example of a Push program and the corresponding execution and metadata traces. Note that there are two things particular to Push program execution that are also reflected in the traces: the whole Push program is kept on the `exec` stack for execution in the beginning, and unpacking a list of instructions is also counted as one execution step.

We first convert the metadata trace, which is composed of the identifiers of the instructions in the execution trace, into a different representation in the following way. A single instruction is represented as `[a:a]`, where `a` is the identifier of that instruction. A group of instructions inside parentheses, with the identifier of the first instruction being `a` and that of the last one being `b`, is represented as `[a:b]`. Additionally, repeating segments in an execution trace are called **instruction-segments** (for example, `(exec_dup (exec_swap 1 2))`), and those in metadata trace are called **identifier-segments** (for example, `[1:4]`).

To extract repeating segments for Reuse, we use the modified metadata trace. In each iteration of the outer loop in Algorithm 2, we search the metadata trace for segments of a certain size. Whenever we find a segment of size s, from the next

$(s + 1)$ items in the trace, we collect the ones which are 'included' in the bigger segment. For example, if `[1:4]` is the segment under consideration, the segments `[1:1]`, `[1:3]`, `[2:4]`, etc. are considered to be 'included' in the bigger segment. Before deleting these smaller segments from the metadata trace, we use them to find segments containing consecutive identifiers as follows. **AllContinuousSeqs()** takes a set of identifier-segments, and outputs those proper subsets which contain consecutive identifiers. For example, if the input is {`[2:2],[3:3],[4:4], [6:8]`}, it will output {`([2:2]), ([3:3]), ([4:4]), ([2:2], [3:3]), ([3:3],[4:4]), ([2:2],[3:3],[4:4]),([6:8])`}. This is in tune with the design principles described in Section 7.1.2, which allows the subsets of repeating segments to be considered as well. Following this procedure, we will have a list of segments at the end of the outer loop. And the segment which appears at least twice in this list will be considered for computing Reuse.

Now, to get instruction-segments for Repetition calculation, we first get unique identifier-segments from the list described above and look for the corresponding instruction-segments in the execution trace. From these segments, the ones which are repeated at least twice are considered for computing Repetition. Note that only the segments containing the same set of instructions with different identifiers are considered for calculating Repetition.

7.2 Using Code Use Metrics to Guide Evolution

Every program generated by a genetic programming system has certain 'Software Quality Features.' One of those features is correctness, measured by errors on a set of test cases. Although correctness—which focuses only on the 'output' of a program, not its 'structure'—might be the most important feature, it is not the only one. There are many other features focusing on the structure of generated programs that might be useful in certain settings. In this chapter, we will term these features 'design

Algorithm 2: Calculating metrics for Push programs

```
Input: execution trace of the program containing instructions;
       metadata trace of the program containing corresponding identifiers;
Result: Reuse and Repetition values
mTrace := modified metadata trace (see Table 7.2 for an example);
len := length of mTrace;
seqs := an empty set;
for i = 1, 2, 3...len do
    for rcs [m:n] in mTrace do
        if n − m == i then
            temp_items := take the next (i + 1) items [m_j, n_j] from mTrace
              provided [m_j, n_j] ≠ [m, n] and m_j ≥ m, and n_j ≤ n ;
            temp_items := sort the items by the first element of each pair;
            seqs := seqs + AllContinuousSeqs(temp_items);
            delete these items from mTrace;
        end
    end
end
seqs_for_reuse = seqs after removing the identifier-segments appearing only
  once;
use Equation 7.3.4 to calculate Reuse;
seqs_for_repetition = unique identifier-segments from seqs;
get the corresponding instruction-segments and their frequencies;
use Equation ?? to calculate Repetition;
```

features' and differentiate them from 'correctness features.' We will use errors on test cases as the correctness feature and the metrics discussed earlier as design features.

In this section, we will explore ways to combine the metrics with errors of individuals to guide the evolution of programs.

7.2.1 Using Design Features during Parent Selection

The procedure to combine design features with error values is different for different parent selection methods. Here, we describe the procedure with lexicase selection [24]. See Section 7.3.3 for more details on the parent selection methods.

What we essentially aim to do here is to have additional pressure to prefer programs with code reuse during parent selection. In each iteration of lexicase selection, the test cases (or the error values on test cases) are shuffled in random order. The design features can be sorted among these error values in one of the following ways:

1. The first option is to shuffle error values and design features together.

2. The second option is to consider design features after error values. What this means is that out of two individuals, one with higher values of design features would be preferred over the other only if they both have the same error values.

3. The third option is to consider design features before error values. What this means is that out of two individuals, one with higher values of design features would always be preferred over the other, irrespective of error values.

We note that the values computed for the metrics are used during selection just like normal error values, but they are not taken into consideration when determining whether a program is a solution to the given problem or not.

Another method of using the metrics to guide the evolution of programs is to filter out individuals with low values of these metrics from the population before the parent selection commences.

7.2.2 Using Design Features during Variation

Code use metrics can also be used to guide the mutation and crossover operators. For example, the values of reuse and repetition can serve as inputs to a given variation operator, and the rate and other parameters of variation can be decided accordingly. Additionally, instead of applying a variation operator uniformly, we can use the reuse and repetition values of the parts of an individual to decide the location of variation.

7.2.3 Experiments and Results

In this section, we describe the experiments that were conducted to investigate the effects of using the metrics as design features. We run our experiments on Clojush[1] (Clojure implementation of PushGP) which evolves programs in a stack-based programming language called Push. Since calculating these metrics is time-intensive, we decided to use the problem of symbolic regression because of its faster runtime per generation compared to other more complex problems. Specifically, we try to evolve programs to compute the mathematical expression of $(x^3 + 1)^3 + 1$.

Before presenting our experimental set-up, we introduce the concept of autosimplification as a way to remove unnecessary instructions from the programs prior to calculating the metrics.

Autosimplification. Sometimes, a program evolved by a given genetic programming system contains many instructions that do not contribute to its performance on the test cases. In other words, some code elements can be removed from the program without affecting its performance on the test cases. While these unnecessary instructions do not affect the overall error vector of the program, they can reduce the accuracy of the metrics presented in this paper. Hence, we perform two sets of

[1] https://github.com/lspector/Clojush

experiments, one where we calculate the metrics on non-simplified programs, and the other where we calculate them on simplified ones.

In genetic programming, automatic simplification of evolved programs has been shown to improve generalization [20]. It has also been used as a genetic operator [20, 62]. Algorithm 3 gives a simple procedure to automatically simplify a given Push program with a certain number of steps. Although the procedure can produce different simplified programs each time it is run, the prior work shows that it often produces consistent results in practice. For programs written in other languages, similar algorithms can be developed.

Algorithm 3: Auto-simplification of Push programs

```
Input: prog, the program to be simplified;
       numSteps, number of simplification steps;
       ErrorFunction(), the function used for calculating errors;

errVector := ErrorFunction(prog) ;
repeat
    rand := a random number between 0 and 1;
    if rand < 0.5 then
        newProg := remove a small number (typically 1 or 2) of random
          instructions from prog;
    else
        newProg := remove a random parenthesis pair from prog;
    end
    newErrVector := ErrorFunction(newProg) ;
    if newErrVector == errVector then
        prog := newProg;
    end
until the numSteps limit is reached;
return prog
```

Experimental Set-up and Results. As discussed in Section 7.2, there are multiple ways of using the code use metrics to incentivize evolving programs to have more reuse and less repetition. In our experiments, we focus on using the metrics during lexicase selection.

Table 7.3: Genetic Programming Parameters.

Parameter	Value
Population size	1000
Number of generations	500
Parent selection algorithm	ϵ-lexicase
Mutation operator	Uniform Mutation by Addition and Deletion
Mutation rate	0.09
Number of runs per condition	30

We follow the procedure laid down in Section 7.2.1. Prior to parent selection, we first compute the errors on a set of test cases. Then to calculate the metrics, we choose a test case randomly from the list of test cases, execute the program on that test case, and then use the execution trace so obtained to compute the metrics. During lexicase selection, we shuffle the errors and metrics together for every selection event. In addition to individuals with low errors, we prefer the ones with high values of Reuse and low values of Repetition.

We perform experiments on the same problem under four different conditions with 30 runs for each condition: without using any metrics, using only the reuse metric, using only the repetition metric, and using both metrics. In these experiments, the metrics are calculated on non-simplified programs. The parameters used during the runs are listed in Table 7.7. To deal with floating-point numbers, we use a version of lexicase selection called ϵ-lexicase selection [32]. We do not perform crossover and instead use UMAD (Uniform Mutation by Addition and Deletion)[21] mutation operator. During evolution, the programs have access to all the instructions that operate on float and execution stacks.

The results are given in Table 7.4. The number of successes is the number of runs out of 30 that evolved a successful program passing all the test cases. The size of a Push program is the total number of instructions and parenthesis pairs present in it. In addition to presenting the average size of successful programs, we also show the average size of successful programs after simplification (number of simplification steps being 5000) so as give a sense of the proportion of nonessential instructions in them. To test statistically significant differences, we used pairwise comparisons for proportions for success rates and the Mann-Whitney-Wilcoxon Test for the sizes of successful programs. From the table, it is clear that using reuse as a design feature does not affect the average size of the successful programs. However, when we use the repetition metric either separately or with reuse, we get significantly larger programs. Using both metrics also leads to a significant decrease in the success rate. The increase in size might be due to the fact that, in order to have a low value of repetition, the programs can often have useless instructions, which basically increases the denominator in Equation 7.6. This is well supported by the last two rows of the table, where the difference between the sizes of programs before and after simplification is very large.

Table 7.4: Using metrics as design features. The metrics have been calculated on non-simplified programs. The underline indicates that the value is significantly different from the corresponding value in the no-intervention case.

	Number of successes	Avg. size of successful programs	Avg. size of successful programs after simplification
No intervention	12	43.08	20.42
Reuse only	5	48.60	12.00
Repetition only	5	68.80	19.00
Both metrics	4	66.75	18.00

To restrain the metrics from preferring programs with more nonessential instructions, we used the concept of autosimplification as discussed in Section 7.2.3. In other words, prior to calculating the metrics, we simplified the program with the number of simplification steps being 50. Accordingly, we performed 30 runs for each of the conditions mentioned earlier. The results are given in Table 7.5. The success rate improves in all cases. Although we get smaller programs while using the reuse metric now, the average program size is still significantly large compared to the no-intervention case whenever the repetition metric is used. In future work, therefore, we can either explore increasing the number of simplification steps or use some other mechanism to rein in the increase in the program size when using the repetition metric.

Although the average size of the successful programs after simplification while using the repetition metric is significantly less than the corresponding value for no-intervention case, the performance in terms of the number of successes and the average size of the non-simplified programs is actually worse.

Table 7.5: Using metrics as design features. The metrics have been calculated on simplified programs, with the number of simplification steps being 50. The underline indicates that the value is significantly different from the corresponding value in the no-intervention case.

	Number of successes	Average size of successful programs	Average size of successful programs after simplification
No intervention	12	43.08	20.42
Reuse only	9	36.44	17.78
Repetition only	10	<u>70.20</u>	<u>10.60</u>
Both metrics	10	<u>59.10</u>	12.80

From the results described in Tables 7.4 and 7.5, it can be concluded that the best results are obtained by computing the reuse metric on simplified programs and using it as a design feature in lexicase selection. These experiments show how the code use metrics can be used in the evolutionary framework.

7.3 Relationships between Parent Selection Methods, Looping Constructs, and Success Rate

In Evolutionary Computation techniques, including genetic programming, certain parent selection algorithms are used to select individuals from the current generation, based on some criteria, to be used as parents for the next generation. Although, in some cases, other factors related to individual programs such as size, novelty, etc. can be used, in the most basic and commonly used versions, these parent selection methods select individuals on the basis of their errors on a number of fitness cases. This leads to differences in the overall fitness of the population during evolution for different selection methods. Since fitness is affected by the presence or absence of certain programming constructs, such as loops, conditionals, etc., in the individuals, these selection methods indirectly affect certain properties of the individuals in the population as well.

In this section, we study one such property, the utilization of loops by evolving individuals, which we calculate using a measure of code reuse described later in the section. Specifically, we investigate whether the selection methods which lead to high success rates on software synthesis problems also lead to a high number of individuals with loops in the population and why. To do this, we conduct and analyze the results of experiments on benchmark software synthesis problems.

In the following subsections, we first describe some related work. Then, in subsections 7.3.2, 7.3.3, and 7.3.4, we describe the genetic programming system in which our experiments were conducted, the parent selection methods that we used, and the measure of code reuse that underlies our analysis of the presence of loops, respectively. The details of our experiments and their results are described in Sections 7.3.5 and 7.4.

7.3.1 Related Work

The structure of individual programs in evolving populations has been studied in a variety of contexts. Li and Ciesielski [35] examined the usefulness of explicit loops and the patterns captured by those loops in a couple of problems in a tree-based genetic programming setting. Trends in the module usage in Grammatical Evolution (GE)—

for example, the number of individuals in the population at a given generation that is using a certain module and the average fitness for such individuals—are studied in [58]. The dynamics of the evolving populations in terms of various structural and behavioral measures of diversity has been studied in [6], where the authors found that some diversity measures, such as the edit distance metric, have a strong correlation with the fitness of a given population, while others have a weak one. The authors, however, cautioned against drawing causal inferences from this work.

Lexicase selection, which we describe in detail later in Section 7.3.3, selects a parent through successively filtering the population of individuals based on their performance on a set of shuffled test cases. Lexicase selection has been compared with other parent selection methods in various settings. It leads to better success rates on multiple problems like designing digital circuits, counting words in files, and various other software synthesis problems, when compared to tournament selection and implicit fitness sharing [23, 24]. It, however, usually takes more time to process a generation with lexicase selection than other selection methods [24]. Metevier et al. [37] studies the performance of lexicase selection in a non-genetic-programming context and compare it against tournament and fitness-proportionate selection algorithms in solving Boolean constraint satisfaction problems. The performance measures used for the comparison were: the success rate, the number of generations used to find a solution, and structural diversity. Lexicase selection has also been compared with tournament selection in terms of its ability to select specialists, i.e., those individuals in the population that have high errors on some cases while low errors on others [22]. The ability to select specialists seems to be one of the reasons why lexicase performs much better than many other selection methods in terms of the success rate on multiple software synthesis problems.

7.3.2 PushGP and Looping instructions

For the experiments presented in this section, we use the genetic programming system called PushGP. It uses linear genomes, which are translated to Push programs for error testing during evolution. In Push, every data type has a separate stack. Instructions in Push can take their inputs from and place their outputs on different stacks. A program is basically a list, possibly nested, of instructions and literals. Instructions are executed on a dedicated stack called the `exec` stack. Since code itself is considered a data type, it can also be manipulated using particular instructions. Instructions come in different forms: simple functions that act on their inputs and execute only once, conditionals that can execute one set of instructions or the other depending on the values of other inputs, and looping instructions that can execute a set of instructions multiple times. In this section, we will focus more on the latter type of instructions. Some examples of looping instructions[2] in PushGP are:

- `exec_dup`: The next code chunk (a single instruction or a group of instructions inside parenthesis) on the `exec` stack is duplicated, which leads to that code chunk getting executed twice unless executing the chunk the first time prevents the execution of the same chunk the second time.

- `exec_do*range`: The next code chunk (a single instruction or a group of instructions inside parenthesis) on the `exec` stack is copied n times, where n is the difference between the first two items on the `integer` stack. This causes the code chunk to be executed n times unless executing one copy of the code chunk prevents the execution of the other copies.

[2]Other looping instructions are: `exec_dup_times`, `exec_dup_items`, `exec_yankdup`, `exec_do*count`, `exec_do*times`, `exec_while`, `exec_do*while`, `exec_s`, `exec_y`, `exec_do*vector_integer`, `exec_do*vector_float`, `exec_do*vector_boolean`, `exec_do*vector_string`.

Since these looping instructions are open-ended, i.e., the number of iterations is not defined in advance and is dependent on the data on other stacks, the programs can sometimes have infinite loops. In order to prevent that, PushGP sets some upper limit on the number of instruction evaluations that can be performed while executing a given program. Usually, this limit is set by the user in advance and is different for different problems. If this limit is exceeded, the program terminates, and the remaining instructions are not executed. Depending on whether a program gets executed within the computational limits set by the system or terminated by the system if it does not, it can be categorized either as a terminating program or a non-terminating program.

Note that although we use PushGP in our experiments in this paper, the conclusions we draw from the results will be applicable to any linear genetic programming system employing at least some of the looping constructs mentioned above.

7.3.3 Parent Selection Methods

Going from one generation to the next involves selecting *promising* individuals from the current generation that can serve as parents for the next one and then using these selected parents as inputs to genetic operators that produce offspring. Such promising individuals are not always those that are best with respect to the total error aggregated across multiple test cases. In many common settings, individuals with total error values that are not particularly good can also get selected as parents in order to maintain diversity and novelty in the population. Various selection methods differ in the way they consider which individuals are promising and which are not.

In this section, we experiment with three commonly used selection methods from the genetic programming literature. The details of these methods are given below.

Lexicase Selection. The detailed procedure to select a parent in this method is given in Algorithm 4 and described as follows. For each selection event, all the test cases are ordered randomly, and all individuals in the population are initially included in a pool of potential parents. Then, for the first test case, the individuals that are best on that test case are kept, and the rest are removed from the pool of potential parents. Note that being *best on a test case* essentially means having the lowest error on that test case among all the individuals in the current pool. The second test case is then considered, and again, all but those that are best on that test case are removed from the pool. The process is repeated until only one individual remains in the pool, or until all of the test cases have been considered. In the latter case, one individual is randomly chosen from the remaining pool and returned as the selected parent. Note that the lexicase selection algorithm does not aggregate errors on the test cases into a single value and instead considers them one by one.

Algorithm 4: Lexicase Selection.

```
for each selection event do
    candidates = individuals in the population ;
    list_of_cases = random_ordering(cases) ;
    for t in list_of_cases do
        candidates = individual in candidates that are best on case t;
        if candidates contains a single individual then
            return the single individual in candidates
        end
    end
    return a randomly chosen individual from candidates
end
```

Tournament Selection. In tournament selection with tournament size t, for every selection event, t individuals are randomly chosen from the population. Then, the individual with the least total error from the set of chosen individuals is returned as the selected parent.

Fitness-proportionate Selection. In fitness-proportionate selection, each individual in the population has a probability of being selected that is proportional to its fitness, which is derived from the total error, aggregated over test cases. If f_i is the

Algorithm 5: Tournament Selection with tournament size t.

for *each selection event* **do**
  candidates = randomly choose t individuals from the population;
  return the individual with the lowest total error from candidates;
end

fitness of an individual, then the probability that it will be selected as a parent in any particular selection event is $\frac{f_i}{\sum_{j=1}^{n} f_j}$. To avoid division by zero, the fitness of an individual can be defined as $\frac{1}{1+totalError_i}$, where $totalError_i$ is the sum of errors on all the test case for the individual i.

7.3.4 Individual with Loops

In order to analyze the effect of parent selection methods on the occurrence of loops in programs, it is necessary to determine whether a given program executes any loops. The simple presence of looping instructions in Push programs does not necessarily mean that those looping instructions will be executed. Other programming constructs in the program, such as conditional instructions, can make the execution of loops conditionally dependent on the execution context. For this reason, and to enable an analysis of the *amount* of looping performed by a program in future work (though not here), we make use of a measure of code *reuse*.

The procedure that we use for calculating the amount of reuse in a given program is given in the previous sections. It takes into consideration the frequency and length of *modules* in program execution traces. Briefly, the procedure is as follows:

1. Assign an identifier to every instruction in the program. Usually, this identifier is the location or index of the instruction in that program.

2. Execute the program and obtain the execution trace by collecting the identifiers of the instructions that are being executed.

3. Designate as modules the instructions and the groups of instructions that appear more than once in the trace.

4. Calculate the amount of reuse using the following equation:

$$Reuse = \frac{\sum_{i=1}^{m} l_i \cdot 2^{f_i}}{2^u}, \tag{7.7}$$

where there are m modules, and module i has a length l_i and appears f_i times in the execution trace. The length of a module is defined as the number of instructions and literals, plus the number of parenthesis pairs.

There can be two kinds of constructs that can lead to a code chunk getting repeated more than once:

- Loop-like: In this type of construct, the body of the loop is executed for a certain number of iterations.

- Function-like: In this type of construct, a code chunk is labeled with an identifier, and the program uses this identifier to call the code chunk multiple times.

In the configuration that we use in this paper, we do not include function-like constructs in the instruction set available to the programs. Therefore, the metric of Equation 7.7 will only calculate the extent to which loop-like constructs are present in the programs. Note that we identify the loops only from the execution trace and not from the way those instructions are defined in the system. Consequently, if the body of a loop executes only once, it will not be considered while calculating the metric using Equation 7.7. Additionally, the procedure takes into account the explicit loops, i.e., the loops which operate on the `exec` stack. Other implicit looping instructions, such as `integer_dup`, which duplicates the item on the `integer` stack, won't have any effect.

Since we are only interested in whether a given program contains loops, we use a modified version of the procedure described above. We look at the execution trace to see if there are any repeating sequences of identifiers. If such a repetition is present, we determine that the program contains loops. For each test case, we have a separate execution trace. But considering the computational costs associated with calculating the reuse metric, in our experiments, we use five execution traces obtained by running the program on five randomly chosen test cases. The number of traces to be used in the reuse calculation is somewhat arbitrary, but this number will not affect the general trend of the data presented in this paper. By the term *individual with loops*, we mean an individual program that has repeating sequences in at least one of the five execution traces used to calculate the reuse metric.

7.3.5 Experiments

As mentioned above, we run our experiments in Clojush, and the key genetic programming parameters for our runs are given in Table 7.7. Other parameters not mentioned here have the standard values used in the literature. For tournament selection, we use a tournament size of 7, a value commonly used in the literature for software synthesis problems. We do not use crossover but instead, use a mutation operator called UMAD (Uniform Mutation by Addition and Deletion). While the absence of crossover may be surprising to some, this choice was driven by the fact that UMAD, without crossover, has thus far produced the highest success rates on the problems studied in the experiments presented here [21]. Additionally, in preliminary experiments, it was found that using crossover in addition to mutation does not change the conclusions drawn from the results presented in this paper, although it may change the exact numbers.

We also do not use any elitist mechanism, which means that all the individuals that proceed to the next generation are selected by a given parent selection method, and all the selected individuals go through the process of genetic variation.

For our experiments, we choose a subset of problems from the General Program Synthesis Benchmark Suite [23], keeping in mind the variety of input and output data types they employ and the state-of-the-art success rate on them. In Table 7.6, in addition to the problem descriptions, we also provide information about the number of training and test cases used during evolution and how the errors are calculated for each of the problems.

Note that while evaluating the fitness of a program for the purpose of selection as a parent, we only consider its errors on a given set of training cases. We do not take into account other measures of quality like program size, complexity, etc.

7.3.6 Number of individuals with loops and Selection Methods

In this set of experiments, we investigate the effect of parent selection methods on the number of individuals with loops in the population for each generation, and whether this number is related to the success rates for different selection methods. Specifically, we analyze how the number of individuals with loops changes across generations for different selection methods across multiple runs.

Figure 7.1 shows the number of individuals with loops across generations for different problems. Each line in the plot represents a single run, and the incomplete lines represent runs that were able to find a solution before hitting the generation limit of 300. If a particular run finds a solution that satisfies all the training cases, we stop the run at that generation. Clearly, for lexicase selection, the lines are concentrated in the upper half of the respective plots. For tournament selection, the lines are all over the place, while for fitness-proportionate selection, the lines are firmly in the lower half of the plots.

Table 7.6: Benchmark problems used in the experiments. The second row for each of the problems presents the number of training and test cases used during evolution, and the third row presents the procedure to calculate the error for a given program for that particular problem.

Last Index of Zero	Description	Given a vector of integers, at least one of which is 0, return the index of the last occurrence of 0 in the vector.
	Training and test split	150, 1000
	Error calculation	Absolute difference between the program output and the correct output.
Digits	Description	Given an integer, print that integer's digits each on their own line starting with the least significant digit. A negative integer should have the negative sign printed before the most significant digit.
	Training and test split	100, 1000
	Error calculation	Levenshtein distance between the program output and the correct output.
Compare String Lengths	Description	Compare String Lengths: Given three strings $n1$, $n2$, and $n3$, return `true` if $length(n1) < length(n2) < length(n3)$, and `false` otherwise.
	Training and test split	100, 1000
	Error calculation	0 if the program output and the correct output is same, 1 otherwise.
Median	Description	Given 3 integers, print their median.
	Training and test split	100, 1000
	Error calculation	0 if the program output and the correct output is same, 1 otherwise.

Table 7.7: Genetic Programming Parameters.

Parameter	Value
Population size	1000
Number of generations	300
Parent selection algorithms	Lexicase, Tournament, Fitness-proportionate
Mutation operator	UMAD (Uniform Mutation by Addition and Deletion)
Mutation rate	0.09
Genome Representation	Plushy
Number of runs condition	50

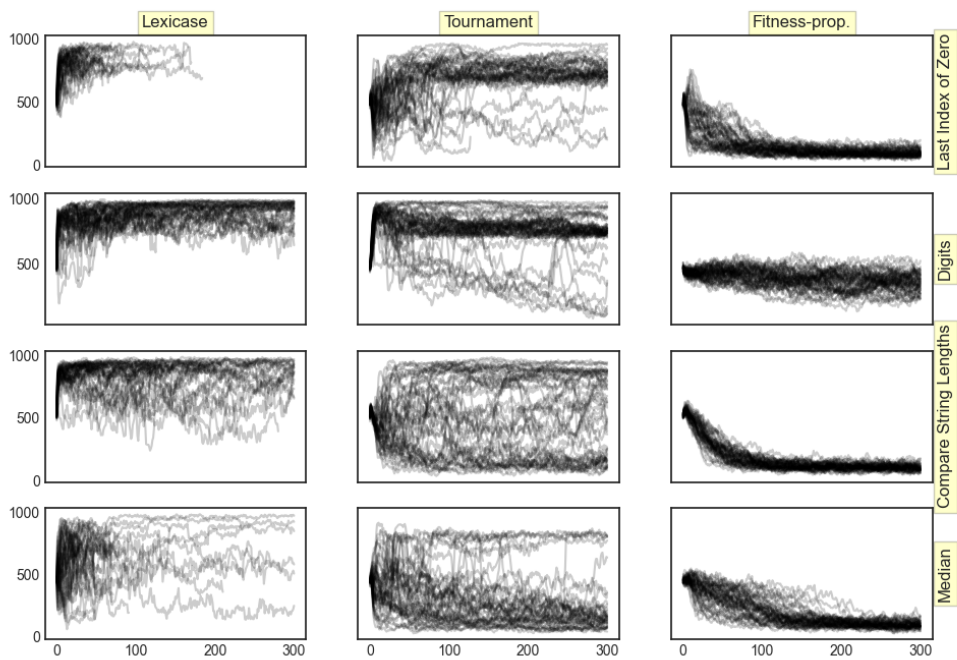


Figure 7.1: Number of individuals with loops (y-axis) for each generation (x-axis). Each line represents a single run, and the incomplete lines represent runs that succeeded in finding a solution before hitting the generation limit of 300.

Table 7.8: Average number of individuals with loops and number of successes, out of 50, for various problems under different selection methods. Boldfaced numbers in each row are significantly more than the other numbers in the same row. We use the pairwise chi-square test at 0.05 significance level.

Problem		Lexicase	Tournament	Fitness-prop.
Last Index of Zero	Inds. with loops	**793.4**	635.13	152.16
	Successes	**37**	14	0
Digits	Inds. with loops	**868.69**	743.00	389.49
	Successes	5	0	0
Compare String Lengths	Inds. with loops	**823.61**	510.21	170.68
	Successes	**18**	1	0
Median	Inds. with loops	**648.7**	322.86	189.23
	Successes	**34**	9	0

Another way to look at the same data is to examine the number of individuals with loops in the population, averaged over all generations across all runs. We calculate this number using the following procedure. We first calculate this number for every run: sum the number of individuals with loops and divide it by the number of generations in the run. For runs that did not produce a solution, the number of generations to be used in the average is 300; otherwise, the number can be less than 300. Then this number is averaged over all the runs and reported. The results are presented in Table 7.8. The numbers confirm what we saw in the plots in Figure 7.1: lexicase selection selects more individuals with loops, followed by tournament and fitness-proportionate selection.

This data leads us to ask why lexicase selection has the highest number of individuals containing loops among the selection methods considered here. To answer this question, we first look at the error profile of the individuals that execute loops using the following procedure. Since we do not use crossover and use only mutation, we need to select 1000 individuals every generation to act as parents for the next generation. We sample 100 individuals out of all the individuals selected by the selection methods for every generation. Therefore, for the runs that reach the generation

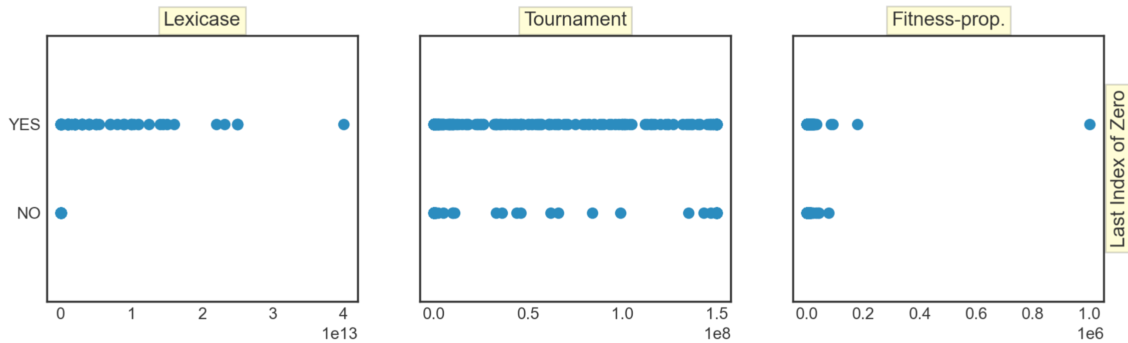


Figure 7.2: Total error values of individuals (x-axis) and whether or not they execute loops (y-axis). Note that the x-axis values for all the subplots are different.

limit of 300, we will have 30000 data points. Otherwise, we will have less. We collect the individuals from all the generations across all the runs. Next, we look at the reuse and total error values of these individuals and figure out the difference between the individuals having loops and those without loops. We present the scatter plot of these values in Figure 7.2 for one of the problems in our set of benchmark problems, i.e., Last Index of Zero. We chose Last Index of Zero as a representative problem because of its moderate success rate, different input and output data types, etc. On the y-axis, we have an indication of whether or not the individual executed loops. On the x-axis, we have the error values.

Clearly, more often than not, individuals with loops have higher total errors than the ones without loops. Now, since lexicase selection selects more individuals with loops than other selection methods, that would imply that lexicase selection selects more individuals with high total error than other selection methods. But why is that? Some of these individuals with a high total error are specialists in the sense that they have low errors on some cases but a very high error on others. Lexicase selection has been shown to select such specialists more frequently than other selection methods [22].

In other words, lexicase selection is more tolerant of high total errors than other selection algorithms. We use a simple example to explain this. Table 7.9 shows three

Table 7.9: An example scenario containing three individuals.

	t1	t2	t3	total error
ind 1	0	0	1000	1000
ind 2	100	100	100	300
ind 3	300	400	301	1001

individuals with errors on three test cases: t1, t2, and t3. Ind 1 seems to be the most desirable since it has zero error on most cases. Lexicase selection will most likely select ind 1, i.e., if t1 or t2 comes first in the random order of the test cases, whereas fitness-proportionate selection will most likely not select it because of the high value of total error. Tournament selection might select ind 1 if it gets selected in a tournament with ind 3.

To summarize, sometimes, the introduction of loops in the program through the genetic operators can lead to infinite loops for some or all of the test cases. Since such programs are terminated by the system abnormally, they tend to have very high total error. But if such loops are beneficial for some test cases, in the sense that the individual has the best errors (i.e., the lowest errors) on these test cases in the population, then lexicase selection will be able to select such individuals with a high probability. Tournament and fitness-proportionate selection, on the other hand, will be less likely to select such individuals.

7.3.7 Number of individuals with loops and Success rate

In Table 7.8, we also report the number of successes, out of 50 runs, for all problems across all conditions. We also use the pairwise chi-square test at 0.05 significance level to test the statistical significance. A run is considered as successful if it evolves a solution that produces zero error on all the cases from the training set used during evolution and on another held-out test set. We also simplify the programs according to the procedure described in [20], before running them on the test set, since simplification often leads to better generalization. Clearly, the average number of individuals with loops is highly correlated with the number of successes that a selection algorithm produces; lexicase selection has significantly more successes and a higher average number of individuals containing loops than other selection methods.

Does this correlation indicate causation? We investigate this issue by asking the following questions:

1. Are high numbers of programs containing loops in a population essential for achieving high success rates?

2. Are high numbers of programs containing loops in a population essential for lexicase selection to out-perform the other selection methods?

We answer these questions by experimenting with the selection methods under two conditions: one, where the programs do not have access to the looping instructions during evolution; two, where the selection methods are prevented from being able to select individuals with loops.

With no looping instructions. We run experiments where we remove all the looping instructions from the instruction set available to the programs during evolution. In this condition, no looping instructions are added either during initialization or while applying the mutation operator. Note that only the looping instructions which operate on the items on the `exec` stacks are excluded; all other instructions remain as they were. The results are given in Table 7.10. The success rate decreases significantly for most problems under most conditions (compared to the success rate presented in Table 7.8). This implies that having access to the looping instructions during evolution is essential to achieving a high success rate for the software synthesis problems across all selection methods.

With no selection for individuals with loops. In another set of experiments, we prevent the selection methods from selecting individuals with loops to see whether

Table 7.10: Number of successes, out of 50, for various problems when looping instructions were not included in the instruction set.

Problem	Lexicase	Tournament	Fitness-prop.
Last Index of Zero	0	0	0
Digits	11	0	0
Compare String Lengths	19	0	0
Median	7	4	0

there is any change in the performance. In this condition, before each selection event, we remove the individuals with loops from the candidate population. The results are given in Table 7.11, which indicate that there is a general decrease in the performance of all the selection methods for all the problems.

From the results shown in Tables 7.10 and 7.11, we can answer the questions we had asked at the beginning of Section 7.3.7:

1. Using looping instructions during evolution and consequently having individuals which execute loops is crucial for achieving high success rates for all selection methods for most of the problems.

2. Lexicase selection performs the best among the selection methods under all the conditions studied here. While its success rate decreases, sometimes significantly, when the programs do not have access to looping instructions or when the system is not able to select individuals with loops, it still performs better than other methods. The implication seems to be that lexicase selection is still able to select specialists, although not as many, when it is subjected to the conditions described above.

7.4 Discussion

Looping instructions appear to be essential for the high success rate on the problems studied here that are taken from the General Program Synthesis Benchmark

Table 7.11: Number of successes, out of 50, for various problems when the selection methods were not able to select individuals with loops.

Problem	lexicase	Tournament	Fitness-prop.
Last Index of Zero	26	1	0
Digits	0	0	0
Compare String Lengths	18	1	0
Median	35	2	0

Suite [23]. This is evident from the data presented in Table 7.10. Although we experimented with only a few problems, the same might be true for other problems as well.

Unfortunately, however, looping instructions can be *dangerous* in the sense that they may be more likely than other instructions to lead to high errors for some test cases. That said, some such individuals may be specialists in the sense that they might have very low errors on some cases while having high errors on others.

Parent selection methods vary in the extent to which they permit the selection of specialists, i.e., the individuals with low errors on some cases and high errors on others. Previous work suggests that support for the selection of specialists is at least partly responsible for the relatively higher success rates obtained when using lexicase selection [22].

The results presented in this work, shown in Figure 7.1 and Table 7.8, show that support for the selection of specialists is also associated with the use of loops in evolving populations, with the most specialist-supporting method (lexicase selection) producing populations with the most loops, the least specialist-supporting method (fitness-proportionate selection) producing populations with the fewest loops, and the method with intermediate support for selecting specialists (tournament selection) producing populations with an intermediate number of loops.

The experiments presented here cannot fully explain the causal interactions among the factors that we have considered. However, they suggest that it is important to allow for the selection of specialists when evolving programs that make use of loops. This would explain both the larger number of loops and the higher success rates produced by lexicase selection.

7.5 Conclusions and Future Work

In this chapter, we first discussed a set of code use metrics—Reuse and Repetition—that measure different aspects of frequently used code segments by the evolving programs. We then presented two ways in which these code metrics can be used in the evolution of programs: to evolve compact programs and to analyze the effect of parent selection algorithms on the programs that use loops.

First, we presented multiple schemes of using code use metrics in the framework of evolving programs. We demonstrated one of these methods by computing the metrics on Push programs and using them as design features in lexicase selection. We presented some preliminary results which show that using the reuse metric gives us more compact successful programs. We use symbolic regression in the first set of experiments, which is very easy to solve and might not even benefit from using these metrics. Therefore, as future work, we need to run the same set of experiments on more complex problems like the ones from the benchmark suite of [23]. We can further optimize the whole procedure to calculate the metrics and also explore other methods of using them, as mentioned in Sections 7.2.1 and 7.2.2.

Second, we studied the interactions between different selection methods, success rates, and the number of individuals in the evolving populations which make use of loops. We found that lexicase selection has the highest number of individuals which execute loops in the population, followed by tournament and fitness-proportionate selection methods. Looking closely, we discover that these individuals with loops often have high values of total error. Since some of these individuals might be specialists which have very low errors on some cases but can have very high total errors, lexicase selection is able to select them, while other selection methods are not likely to do so.

The number of individuals with loops is highly correlated with the success rates on multiple problems for the selection methods. And although the success rate decreases when the evolution does not have access to the looping instructions or when the selection algorithms are prevented from selecting individuals with loops, lexicase selection still manages to give the best performance among the selection methods considered here.

In the second set of experiments, we only consider whether or not an individual makes use of loops. Future work would look into how the amount of reuse, as discussed in Section 7.3.4, is related to different selection methods and how it affects the success rates on software synthesis problems. More research is also needed to examine whether modifying the tournament and fitness proportionate selection methods to make them select more individuals making use of loops can improve their performance with respect to the success rate. Replicating the experiments in this paper for other genetic programming systems like tree-based, grammar-guided, etc., can also be looked into in the future.

While the metrics presented in this chapter focus only on the structure of code segments, new metrics that take into account the usefulness of those segments in solving different test cases can also be added to the metrics suite.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

In this dissertation, we explored various aspects of modularity in the field of genetic programming. First, we presented a framework called Genetic Learning by Extraction and Absorption (GLEAM) that can be used with most, if not all, of the genetic programming systems to evolve modular programs. We demonstrated its efficacy by implementing it in the PushGP system. We observe that using GLEAM with PushGP leads to significant improvements in the success rate for various software synthesis benchmark problems. These problems are especially challenging because they use multiple data types in inputs and outputs, and require the use of programming constructs such as loops, conditionals, etc.

Second, we proposed a set of hypotheses for why labeled modules might be useful for programs that evolve and tested them in the context of GLEAM with PushGP. Among all the hypotheses that were tested, one seems to explain the usefulness of modules, at least in the case of GLEAM: programs use modules to reuse code segments from other programs.

Third, we proposed two code use metrics—Reuse and Repetition—that measure how frequently the code segments of different sizes have been used in a given program. We showed that using the reuse metric during selection can reduce the size of solutions. We also showed that different parent selection algorithms affect the number of individuals in the evolving population. The parent selection algorithm which leads to the highest success rate among all the selection methods considered—

lexicase selection—also leads to the highest number of individuals with non-zero reuse in them.

In terms of future work, implementing GLEAM for other genetic programming systems seems like a natural extension. We have already delineated the idea of how it can be implemented in Grammatical Evolution. But there are still other systems like Cartesian Genetic Programming, etc., for which GLEAM can prove useful. Even for PushGP, the search for better parameter settings than those used in this dissertation can be carried out. Looking closely at the results of experiments, we notice that some problems seem to benefit a lot more than others from GLEAM. More research is needed to figure out which characteristics of the problems make them more compatible with GLEAM or any other modularity-inducing systems.

We tested the hypotheses related to the usefulness of modules in evolving programs for GLEAM. Some, if not all, of these hypotheses can be tested for other genetic programming systems in which the evolution of modular programs leads to an improvement in the success rate for some problems. In previous chapters, we have mentioned various reasons for why it is difficult to evolve modular programs. These can be tested experimentally, either with GLEAM or some other similar system in the future.

The metrics that we introduced here take into account only the 'structure' of the programs and code segments, i.e., how often different code segments appear or are used by the program. Other metrics that take into account the 'semantics' of these code segments, or the changes they make on the state of the programs during execution, can be introduced. The code segments which seem to be important or useful to the programs, inferred either by their frequency, semantics, or any other similar measure, may be used as new modules during the extraction phase of GLEAM. Whether doing so will further improve the performance of GLEAM can be taken up as future work.

BIBLIOGRAPHY

[1] Angeline, Peter J, and Pollack, Jordan B. The evolutionary induction of subroutines. In *Proceedings of the fourteenth annual conference of the cognitive science society* (1992), Bloomington, Indiana, pp. 236–241.

[2] Balog, Matej, Gaunt, Alexander L, Brockschmidt, Marc, Nowozin, Sebastian, and Tarlow, Daniel. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[3] Banzhaf, Wolfgang, Banscherus, Dirk, and Dittrich, Peter. Hierarchical genetic programming using local modules. In *Unifying Themes in Complex Systems*. CRC Press, 2018, pp. 321–330.

[4] Bonner, John Tyler. *The evolution of culture in animals*. Princeton University Press, 2018.

[5] Brown, Tom, Mann, Benjamin, Ryder, Nick, Subbiah, Melanie, Kaplan, Jared D, Dhariwal, Prafulla, Neelakantan, Arvind, Shyam, Pranav, Sastry, Girish, Askell, Amanda, et al. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[6] Burke, Edmund K, Gustafson, Steven, and Kendall, Graham. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation 8*, 1 (2004), 47–62.

[7] Callebaut, W, and Rasskin-Gutman, D. *Modularity: understanding the development and evolution of natural complex systems*. MIT press Cambridge, MA, 2005.

[8] Chen, Mark, Tworek, Jerry, Jun, Heewoo, Yuan, Qiming, Pinto, Henrique Ponde de Oliveira, Kaplan, Jared, Edwards, Harri, Burda, Yuri, Joseph, Nicholas, Brockman, Greg, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Clune, Jeff, Mouret, Jean-Baptiste, and Lipson, Hod. The evolutionary origins of modularity. *Proceedings of the Royal Society b: Biological sciences 280*, 1755 (2013), 20122863.

[10] Creanza, Nicole, Kolodny, Oren, and Feldman, Marcus W. Cultural evolutionary theory: How culture evolves and why it matters. *Proceedings of the National Academy of Sciences 114*, 30 (2017), 7782–7789.

[11] Demšar, Janez. Statistical comparisons of classifiers over multiple data sets. *The Journal of Machine learning research 7* (2006), 1–30.

[12] Dhama, Harpal. Quantitative models of cohesion and coupling in software. *Journal of Systems and Software 29*, 1 (1995), 65–74.

[13] Eide, Eric N. Why Use Functions? `https://www.cs.utah.edu/~zachary/computing/lessons/uces-10/uces-10/node11.html`. [Online; accessed 19-Jun-2022].

[14] Ellis, Kevin, Wong, Catherine, Nye, Maxwell, Sablé-Meyer, Mathias, Morales, Lucas, Hewitt, Luke, Cary, Luc, Solar-Lezama, Armando, and Tenenbaum, Joshua B. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation* (2021), pp. 835–850.

[15] Espinosa-Soto, Carlos, and Wagner, Andreas. Specialization can drive the evolution of modularity. *PLoS computational biology 6*, 3 (2010), e1000719.

[16] Gulwani, Sumit. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices 46*, 1 (2011), 317–330.

[17] Gulwani, Sumit, Polozov, Oleksandr, Singh, Rishabh, et al. Program synthesis. *Foundations and Trends® in Programming Languages 4*, 1-2 (2017), 1–119.

[18] Helmuth, Thomas, and Abdelhady, Amr. Benchmarking parent selection for program synthesis by genetic programming. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (2020), pp. 237–238.

[19] Helmuth, Thomas, and Kelly, Peter. PSB2: The second program synthesis benchmark suite. In *2021 Genetic and Evolutionary Computation Conference* (Lille, France, 10-14July 2021), GECCO '21, ACM.

[20] Helmuth, Thomas, McPhee, Nicholas Freitag, Pantridge, Edward, and Spector, Lee. Improving generalization of evolved programs through automatic simplification. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2017), pp. 937–944.

[21] Helmuth, Thomas, McPhee, Nicholas Freitag, and Spector, Lee. Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2018), pp. 1127–1134.

[22] Helmuth, Thomas, Pantridge, Edward, and Spector, Lee. On the importance of specialists for lexicase selection. *Genetic Programming and Evolvable Machines 21*, 3 (2020), 349–373.

[23] Helmuth, Thomas, and Spector, Lee. General program synthesis benchmark suite. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation* (2015), pp. 1039–1046.

[24] Helmuth, Thomas, Spector, Lee, and Matheson, James. Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation 19*, 5 (2014), 630–643.

[25] Keijzer, Maarten, Ryan, Conor, and Cattolico, Mike. Run transferable libraries—learning functional bias in problem domains. In *Genetic and Evolutionary Computation Conference* (2004), Springer, pp. 531–542.

[26] Kelly, Stephen, Newsted, Jacob, Banzhaf, Wolfgang, and Gondro, Cedric. A modular memory framework for time series prediction. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference* (2020), pp. 949–957.

[27] Kernighan, Brian W, and Ritchie, Dennis M. *The C programming language.* Pearson Educación, 1988.

[28] Koza, John R. Human-competitive results produced by genetic programming. *Genetic programming and evolvable machines 11*, 3-4 (2010), 251–284.

[29] Koza, John R, Andre, David, Keane, Martin A, and Bennett III, Forrest H. *Genetic programming III: Darwinian invention and problem solving*, vol. 3. Morgan Kaufmann, 1999.

[30] Koza, John R, and Koza, John R. *Genetic programming: on the programming of computers by means of natural selection*, vol. 1. MIT press, 1992.

[31] Krawiec, Krzysztof, and Wieloch, Bartosz. Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (2009), pp. 995–1002.

[32] La Cava, William, Spector, Lee, and Danai, Kourosh. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016* (2016), pp. 741–748.

[33] Lalejini, Alexander, and Ofria, Charles. Evolving event-driven programs with signalgp. In *Proceedings of the Genetic and Evolutionary Computation Conference* (2018), pp. 1135–1142.

[34] Lalejini, Alexander, and Ofria, Charles. What else is in an evolved name? exploring evolvable specificity with signalgp. In *Genetic Programming Theory and Practice XVI*. Springer, 2019, pp. 103–121.

[35] Li, Xiang, and Ciesielski, Victor. An analysis of explicit loops in genetic programming. In *2005 IEEE Congress on Evolutionary Computation* (2005), vol. 3, IEEE, pp. 2522–2529.

[36] Manna, Zohar, and Waldinger, Richard J. Toward automatic program synthesis. *Communications of the ACM 14*, 3 (1971), 151–165.

[37] Metevier, Blossom, Saini, Anil Kumar, and Spector, Lee. Lexicase selection beyond genetic programming. In *Genetic Programming Theory and Practice XVI*. Springer, 2019, pp. 123–136.

[38] Newman, Mark EJ. Modularity and community structure in networks. *Proceedings of the national academy of sciences 103*, 23 (2006), 8577–8582.

[39] O'Neill, Michael, and Spector, Lee. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines* (2019), 1–12.

[40] Ryan, Conor, Collins, John James, and Neill, Michael O. Grammatical evolution: Evolving programs for an arbitrary language. In *European conference on genetic programming* (1998), Springer, pp. 83–96.

[41] Saini, Anil Kumar, and Spector, Lee. Modularity metrics for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2019), pp. 2056–2059.

[42] Saini, Anil Kumar, and Spector, Lee. Effect of parent selection methods on modularity. In *European Conference on Genetic Programming (Part of EvoStar)* (2020), Springer, pp. 184–194.

[43] Saini, Anil Kumar, and Spector, Lee. Using modularity metrics as design features to guide evolution in genetic programming. *Genetic Programming Theory and Practice XVII* (2020), 165.

[44] Saini, Anil Kumar, and Spector, Lee. Why and when are loops useful in genetic programming? In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion* (2020), pp. 247–248.

[45] Saini, Anil Kumar, and Spector, Lee. Gleam: genetic learning by extraction and absorption of modules. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2021), pp. 263–264.

[46] Saini, Anil Kumar, and Spector, Lee. Relationships between parent selection methods, looping constructs, and success rate in genetic programming. *Genetic Programming and Evolvable Machines* (2021).

[47] Saini, Anil Kumar, and Spector, Lee. Evolving and analyzing modularity with gleam (genetic learning by extraction and absorption of modules). In *Genetic Programming Theory and Practice XVIII*. Springer, 2022, pp. 181–195.

[48] Saini, Anil Kumar, Spector, Lee, and Helmuth, Thomas. Environments with local scopes for modules in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2022), pp. 598–601.

[49] Simon, Herbert A. The architecture of complexity. In *Facets of systems science*. Springer, 1991, pp. 457–476.

[50] Sobania, Dominik, Schweim, Dirk, and Rothlauf, Franz. Recent developments in program synthesis with evolutionary algorithms. *arXiv preprint arXiv:2108.12227* (2021).

[51] Spector, Lee. Evolving control structures with automatically defined macros. In *Working Notes of the AAAI Fall Symposium on Genetic Programming* (1995), pp. 99–105.

[52] Spector, Lee, Harrington, Kyle, and Helmuth, Thomas. Tag-based modularity in tree-based genetic programming. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation* (2012), pp. 815–822.

[53] Spector, Lee, Harrington, Kyle, Martin, Brian, and Helmuth, Thomas. What's in an evolved name? the evolution of modularity via tag-based reference. In *Genetic Programming Theory and Practice IX*. Springer, 2011, pp. 1–16.

[54] Spector, Lee, Klein, Jon, and Keijzer, Maarten. The push3 execution stack and the evolution of control. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation* (2005), pp. 1689–1696.

[55] Spector, Lee, and Luke, Sean. Culture enhances the evolvability of cognition. In *Cognitive Science (CogSci) 1996 Conference Proceedings* (1996).

[56] Spector, Lee, Martin, Brian, Harrington, Kyle, and Helmuth, Thomas. Tag-based modules in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation* (2011), pp. 1419–1426.

[57] Spector, Lee, and Robinson, Alan. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines 3*, 1 (2002), 7–40.

[58] Swafford, John Mark, Hemberg, Erik, O'Neill, Michael, and Brabazon, Anthony. Analyzing module usage in grammatical evolution. In *International Conference on Parallel Problem Solving from Nature* (2012), Springer, pp. 347–356.

[59] Walker, James Alfred, and Miller, Julian Francis. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation 12*, 4 (2008), 397–417.

[60] Wilson, Edwin B. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association 22*, 158 (1927), 209–212.

[61] Zelditch, Miriam L, and Goswami, Anjali. What does modularity mean? *Evolution & Development 23*, 5 (2021), 377–403.

[62] Zhan, Haoxi. A quantitative analysis of the simplification genetic operator. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation* (2014), pp. 1077–1080.