AN ABSTRACT OF THE THESIS OF

Deepthi S Kumar for the degree of Master of Science in Computer Science presented on December 7, 2017.

Title: A Language for Querying Source Code Repositories

Abstract approved: _____

Martin Erwig

The history of a software project plays a vital role in the software development process. Version control systems enable users of a software repository to look at the evolution of the source code, and see the changes that led to newer versions. Currently, version control systems provide commands that can be used to retrieve information about the history, such as, listing different versions, viewing the difference between two arbitrary versions, inspecting changes made in a specific version, and so on. These commands, in one way or the other, require users to first identify the version and then browse through the changes made in it. In this thesis, I present GitQL, a language for querying the history of files that are version-controlled by git. GitQL provides more general queries than git and enables users, for example, to look for specific changes in a file or just retrieve all the changes made to a specific text. GitQL is implemented based on vgrep, a generalization of grep that works on variational strings, which form the formal basis for version-controlled files. Finally, I evaluate the expressiveness of GitQL.

A Language for Querying Source Code Repositories

by

Deepthi S Kumar

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 7, 2017
Commencement June 2018

Master of Science thesis of Deepthi S Kumar presented on December 7, 2017.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Deepthi S Kumar, Author

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Martin Erwig for the continuous support and motivation during the program, and his patience and knowledge that helped me in writing this thesis.

Thank you to Dr. Eric Walkingshaw for helping me troubleshoot the implementation. Finally, I would like to thank my family and friends for supporting me all along.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

Version control systems, also known as revision control systems, are an integral part of the software engineering process. These systems enable users to manage changes made to software repositories consisting of artifacts like source code, usage documents, system specification documents, images, and many other types of files. All the changes made to these artifacts are tracked by creating a new version everytime a change is committed to the repository. For each of these changes, a version control system (VCS) also stores information such as the author's name, time of the commit, reason for the change, and so on. These are called commit metadata. Each of the versions of a file can be retrieved, compared, or restored if required. All the versions of every file in a software repository along with the commit metadata constitute its history.

One of the advantages of using VCS is that the users of a software repository can look at the evolution of the software. Users can see the changes that led to newer versions. For example, when programmers have to enhance an existing feature, they can view changes that were made to develop it, or when the programmers have to understand the change in the behavior of current version from the previous one, they can look at the difference between the two file versions. In the case of unexpected behavior, the programmers can narrow down the part of the code that caused it by identifying what changed between the version that behaved correctly and the current one. Hence, the history of a software repository plays a vital role in its maintenance.

The history is essential even during the development phase when, a programmer tries different implementations and changes the code frequently. A VCS makes this easier using branches. Various implementations can be tracked in different branches and then compared. The programmer can also have all the changes tracked in a single branch, in which case the older changes can be obtained by reverting to a previous version. Therefore, viewing and identifying the changes in a more recent, shorter history helps in the development process.

Git, which is a distributed version control system has gained immense popularity over the last decade. A 2015 survey [21] by StackOverflow shows that 69.3% of the

developers prefer git over other version control systems. Github [13] which offers a web-based version control service uses git and hosts over 55 million projects. Hence, we looked at the history of projects that are version controlled by git.

Querying the history can involve the following.

1 Searching for text inside a file of any version.

2 Looking for the changes between two versions.

3 Filtering the commits based on commit metadata.

Searching for a word or a sentence in a single file is a common task performed by users using many different tools. One can look for exact words or a pattern of words in which case a regular expression is used to specify the pattern. Grep is one such tool in Unix that can search a file using regular expressions.

In a file that is version controlled, searching for text involves looking into a particular version or all the versions of the file. To find a version that consists of a pattern, the user can create a script that runs grep on all the versions and returns the ones that contain the pattern. This brute-force approach is straightforward and easy to implement. However, it is not efficient when there is an enormous number of versions.

An alternative is to use the commands provided by the VCS itself. Since these commands directly work on the internal representation, they are faster than the brute-force approach. In the case of git, commands like `git log`, `git diff`, and `git show` along with various options can be used to query the history. Some of the options enable users to search using regular expressions.

However, in cases where the users want to query the changes made to a particular part of the program, git commands do not suffice.

Consider a scenario where two developers, Jen and Bob are working on a program that is version controlled by git, shown in listing 1.1. The program involves accessing user information from the database. The function `elementOf` parses a list of users and returns true if an arbitrary user exists, otherwise it returns false. This function is used before making an expensive database access to retrieve any data about a user.

```
...
Boolean elementOf(ID userID)
{
 String userID = regexLookUp("*userID",this.userList);
  if(userID == "")
   return False;
  else
   return True;
}
...
User getUserInfo(ID userID)
{
 User user = Null;
 if(!elementOf(userID))
  throw new Exception("User not present");
 else
  user = getUserInfoFromDB(userID);
 return user;
}
...
```

Listing 1.1: Code snippet

Jen is working on a new feature and adds a few methods to this program. Some of the new methods use `elementOf`. Listing 1.2 shows one of the new methods added by Jen. Bob on the other hand is refactoring the existing code. He changes the method name `elementOf` to `isCurrentUser`. However, Bob renames it twice before arriving at this name. First, to `contains` and then to `present`, all of them committed to the repository. Listings 1.3, 1.4, and 1.5 show the differences in each commit that renames the `elementOf` method.

```
...
Link getUserlinks(ID userID1, ID userID2)
{
 User user1 = Null;
 User user2 = Null;
 if(!elementOf(userID1))
  throw new Exception("User %1 not present", userID1);
```

```
 if(!elementOf(userID2))
  throw new Exception("User %1 not present", userID2);

 user1 = getUserInfoFromDB(userID1);
 user2 = getUserInfoFromDB(userID2);
 return getlinks(user1,user2);
}
...
```

Listing 1.2: Code snippet

```
-Boolean elementOf(ID userID)
+Boolean contains(ID userID)
{
        String userID = regexLookUp("*userID",this.userList)
        if(userID == "")
@@ -19,7 +19,7 @@ Boolean elementOf(ID userID)
User getUserInfo(ID userID)
{
        User user = Null;
- if(!elementOf(userID))
+ if(!contains(userID))
                throw new Exception("User not present");
```

Listing 1.3: Renamed to `contains`

```
-Boolean contains(ID userID)
+Boolean present(ID userID)
{
        String userID = regexLookUp("*userID",this.userList)
        if(userID == "")
@@ -19,7 +19,7 @@ Boolean elementOf(ID userID)
User getUserInfo(ID userID)
{
        User user = Null;
- if(!contains(userID))
+ if(!present(userID))
                throw new Exception("User not present");
```

Listing 1.4: Renamed to `present`

```
-Boolean present(ID userID)
+Boolean isCurrentUser(ID userID)
{
        String userID = regexLookUp("*userID",this.userList)
        if(userID == "")
@@ -19,7 +19,7 @@ Boolean elementOf(ID userID)
User getUserInfo(ID userID)
{
        User user = Null;
- if(!present(userID))
+ if(!isCurrentUser(userID))
                throw new Exception("User not present");
```

Listing 1.5: Renamed to `isCurrentUser`

Both Jen and Bob are working on different branches. First, Bob merges his changes with the main branch. After Jen merges her changes, she finds out that the program does not compile successfully. The method `elementOf` no longer exists. There are three different ways to identify the problem.

1 Look for the file that contained `elementOf` and then search through the file that has the same implementation as `elementOf`.

2 Go through all the commits, made after Jen created the branch, one by one.

3 Search for the commit that renamed the method using the command `git log -G'elementOf'`.

Option 1 and 2 are quite tedious if the implementation is complex or if the number of commits are large. The command in option 3 returns commits that have `elementOf` in the added or removed lines. It shows even the commit that added the function for the first time. By going through the commits returned by the command, Jen sees that it was renamed to `contains`. However, that is not the latest name of the method. Jen has to then search again using `git log -G'contains'` to know what `contains` was renamed to and so on until the latest name is known. This may take less time than the other two options but nonetheless is tedious.

Hence there is a need for a language that is simple, expressive, and enables users not just to look for patterns within a single version but also to look for changes in the entire history of a software repository.

In the remainder of this thesis I will present GitQL, in which users can query the history similar to grep by using regular expressions. The important feature, however, is to query specific changes. This is achieved using a feature called *choice patterns*. With a choice pattern, a user can specify a pattern for the text in the previous version and another pattern for the text in the later version, therefore retrieving only those versions that made changes matching the choice pattern. Users can also specify the text in one version and obtain subsequent changes made to it.

In the above example scenario, Jen would query using the choice pattern $d\langle$`elementOf`, `$x`$\rangle$ and obtain all the subsequent changes made to `elementOf`. The changes will be bound to the variable `x`. Choice patterns are explained in detail in further chapters.

In the next chapter, I describe the existing commands in git for querying version history and related works in the area of querying software repositories and source code changes. In chapter 3, I present the grammar of GitQL, a description of the constructs, and example queries. Implementation details of the underlying representation of software repositories and the search engine for GitQL queries are described in chapter 4. Chapter 5 consists of the evaluation of the expressiveness of GitQL.

Chapter 2: Literature Review

In section 2.1, I discuss some of the state-of-the art features in version control systems that enable developers to query the history of software projects. In section 2.2, I outline some of the tools used in mining software repositories. Finally, in section 2.3, I briefly explain program differencing tools and algorithms.

Before proceeding to the existing tools and commands, I will briefly discuss a related application of the choice calculus [10] in the area of editing and viewing variational software. Projectional editing model [26] which is a view-based model for editing programs with lots of variations (in the form of preprocessors like `ifdef` statements) uses the choice calculus representation which is also used to represent version-controlled files as part of GitQL. The choice calculus and its application in GitQL are explained in detail in chapter 3. In this model, when editing a variational program, a partial variational program that consists of a subset of variations from the original variational program is projected to the user. The user then makes changes concerning only the variants that are obtained by the partial variational program. These edits are then propagated back to the original variational program. This way, a user can easily make changes to specific variants of a program instead of having to carefully examine the changes with respect to all the variants. In the case of version control systems, the view is more restricted in that the textual changes are always made with respect to one specific version which is obtained by applying all the previous edits unlike in the projectional editing model where edits can be made on multiple variants at once. These changes are applied to later versions as well unless they are reverted or changed during a merge conflict resolution. In this thesis, we focus on how these edits can be represented using the choice calculus and then queried. We use the choice edit model [9] to represent changes made to version-controlled files.

2.1 Version Control Systems

In this section I first describe some of the functionalities offered by git, and briefly comment on similar features in a number of other version control systems.

Git [12] is a popular tool to manage source code. A repository in git can have a central server from which the developers/collaborators can download the repository along with its historical data and work locally. Each local copy is a version-controlled repository on its own. The changes made can be tracked locally. The developers can incrementally develop and commit the changes to the local repository and then upload it at once to the remote shared server to make it available to other collaborators.

Git enables developers to access historical information without having to access the shared server. This means that while newer changes are being made, users can almost instantly look into the history to obtain useful information about the changes made to the source code. To search the history of a repository, git provides three main commands: `git log`, `git diff`, and `git show`.

The command `git log` can be used to query most of the historical data. For example, finding all the commits made by a certain author (option `--author`), finding all the commits after a specific date (options `--since` and `--after`) or viewing the parents of a commit (option `--parents`).

Users can also search a commit using the commit message with the `--grep` option. For example, `git log --grep="bug fix"` will print all the commits that have `bug fix` in their commit message.

Apart from querying the metadata of commits, git also allows users to query code changes. The command `git log -S`*str* takes a string *str* and retrieves all the commits that added or deleted the string in any file, at any time in the history of the repository. Users can limit the search to a file by specifying the file name. Consider the following scenario:

A function with the name `getDatabaseInfo` was added in commit 10[1] along with a call to the function. Two more function calls were added in subsequent commits, say commit 15 and commit 20. The function definition and function calls were commented out in commit 25. Finally, the function definition along with the function calls were

[1] Commit identifiers in git are hash codes computed using the commit data. To simplify the presentation, I will use integers as commit identifiers.

removed later in commit 30.

Suppose a user wants to know when the function was deleted. The command `git log -SgetDatabaseInfo` will print all the commits that either added, deleted, or modified the string `getDatabaseInfo`. In the above scenario, git will print the commits 10, 15, 20, and 30. The user then looks at the changes made in each of the commits to see all the changes involving the string `getDatabaseInfo` and finds that commit 30 deleted the function. Notice that commit 25 is not printed by the command. This is because no change was made to the specific string `getDatabaseInfo`.

Git provides another option `-G`*regexp* to look for changes made to the source code. It is similar to the `-S` option but takes a regular expression of the string to be searched and prints all the commits that made changes to the lines that contain the string matched by *regexp*. In the above scenario, the command `git log -G"getDatabaseInfo"` prints commits 10, 15, 20, 25, and 30. Commit 25 modifies a line that has `getDatabaseInfo` and therefore is in the output. Again, the user has to manually look through all the commits to identify the commit that deleted the function `getDatabaseInfo`.

Another option is `-L` which can retrieve the history of a block of code. The range of the block is specified using line numbers. For example, `git log -L 18,23:file1.java` will print all the commits that affected the code fragment in lines 18 to 23. The command sometimes gives unexpected results. For example, if there are commits that involve changes made to the border of the specified code fragment like in the given example, if there were changes in lines 17-18 or in lines 23-25, then those commits are not listed in the output.

Users can specify regular expressions instead of the line numbers to match the beginning and end of the code fragment. The regular expressions is matched against the current version of the file in order to detect the code fragment. Hence, lines that were added in previous commits but are not present in the current version cannot be matched using this option.

The command `git diff` shows line-level difference between two commits. Once a user has identified the commits they want to compare, they can use this command to view the difference. It uses the UNIX `diff` tool, explained in section 2.3 on page 17, to show line-based differences between the two versions. Users can also configure it to use different tools.

The command `git show` is similar to `git diff`, in that it shows the changes made

in a commit. It compares the given commit with the previous commit(s) to compute the changes.

In case a user wants to query specific changes made to a particular part of the program, git commands do not suffice. One could argue that this can be achieved using the above commands twice, once for the string before the change and the other for the string after the change. However, this requires a user to go through all the commits returned by the two commands and then reconcile them with each other manually, since both commands look for the string in added and removed lines.

Subversion [24] provides a command `svn blame` to identify the latest commit that affected a line. When used with an older revision number, it displays the latest commit that affected each line until the specified revision number. However, it does not provide commands to track changes through time, i.e., finding out when a particular line was added or deleted, let alone query specific changes.

Darcs [4] stores the changes made in each version as patches unlike git or subversion which store the versions as snapshots of files. Darcs provides commands such as `darcs log` that lists all the versions, `darcs log -v` that lists all the versions along with the source code changes, `darcs annotate` that shows the latest versions that changed each line in a file, and `darcs changes --matches` *pattern* that selects all the patches that contain strings matching *pattern* similar to the git command `git log -S`. However, it does not have commands that can fetch specific patches based on what was before and after a particular patch was applied.

In mercurial [18], the command `hg grep` allows users to look for all the versions of a file that contain strings matched by a pattern. However, the results of the command need further processing in order to know when a string was added or removed. Also, there is no command that lets users to query specific changes.

Similar to Subversion, CVS [3] offers the command `cvs annotate` that prints the latest version that affected each line in the file. There is no command that directly shows when a string was added or removed or that can be used to find specific changes made to the file.

2.2 Mining Software Repositories

In this section I look at some of the tools that mine software repositories. There has been a lot of research on mining software repositories to obtain information about the evolution of the software. Some of these work on an abstract syntax tree (AST) representation and are therefore language dependent while others are language independent. Although GitQL is language independent, I look at AST-based tools to compare querying capabilities with respect to the history of a software repository.

QWALKEKO [22] is a history querying tool for projects in git. QWALKEKO provides a tool set for the researchers in the field of mining software repositories [23]. QWALKEKO is a combination of a graph query language QWAL and a program query language called EKEKO. The history of the software repository is represented using graphs where each node is a commit and contains information such as author, timestamp, commit message along with the modified files. QWAL is used to navigate this graph and specify the revisions that should be used for further querying. Once the version is identified, the source code of that version is checked out and is then queried using EKEKO. EKEKO is a clojure library containing predicates that can be used over the abstract syntax tree of Java programs. QWALKEKO adds a tree differencer called CHANGENODES that allows users to compute and reason over source code changes. It allows the users to query different aspects of a software project.

Source Code Characteristics: The structure and language properties of the source code. For example, to view the version containing a specific method call or to view all the public methods in the class.

Revision Characteristics: To query the metadata of a commit such as the author, commit message and so on.

Temporal Characteristics: Combining the version history and source code to query temporal aspects. For example, to view the commits that changed a method *foo* before it was deleted.

Changes Characteristics: Querying specific source code changes between two version.

We look at how querying in each of the aspect mentioned above can be achieved.

Source Code Characteristics: Users can query the structural, control flow, and data flow relations of the project. The queries that fall into this category are, for example, determining if a method is being called in the program or viewing all the public methods of a class.

This is a query to view all the function calls in a program.

```
(ekeko [?inv] (ast :MethodInvocation ?inv))
```

Here the keyword `MethodInvocation` is used to specify functional calls and `inv` is a variable that binds to the AST node that matches. Variables are prefixed by `?`.

Revision Characteristics: Metadata of a commit such as the author, commit message and so on that are stored in git can be queried. Predicates from EKEKO can be used to specify the commit properties. These predicates are applied to the metadata of each node in the commit graph.

For example, to match a specific author, the predicate `(author ?name)` is used. The variable `name` binds to the author of the commit. This variable can be used later in the query to match against a specific author.

Temporal Characteristics: Querying the temporal characteristics such as parent or descendants of a commit require navigating the graph of commits used by QWALKEKO. The specifications for these queries are made using QWAL, which uses regular path expressions [5]. Regular path expressions are like regular expression except that they match paths in a graph. Using this concept, QWAL matches the revisions for which the predicates hold. The general structure of a QWAL query is as follows:

```
(qwal graph begin ?end [vars] goals)
```

where `graph` is the instance of the commit graph constructed from the git history, `begin` is the version from where the search needs to begin, `vars` is the list of variables used in the query, goals are the set of predicates that must hold in the current node of the query, and the variable `end` is bound to the last commit in the path on which the predicates hold.

For example, here is a QWAL query to find all the commits that add a file.

```
(qwal graph root ?end []
  (q=>)
  (in-git-info [curr]
    (fileinfo|add ?info curr)))
```

The query loops over all the commits and matches the current commit if it has added new files. The variable `curr` refers to the current node of the graph, and `q=>` is a way to loop over the commits. It moves the current version to one of its successors. Similarly, `q<=` moves the current version to one of its predecessors. The variable `info` binds to the newly added file. The keyword `in-git-info` specifies that the predicate operates on the metadata whereas the keyword `in-source-code` is used to specify predicates on the source code. When `in-source-code` is used, every version of the project is checked out.

Change Characteristics: To query specific source code changes between two versions of a program, QWALKEKO uses CHANGENODES, which takes two AST nodes and returns a minimal edit script required to transform the first AST to the second one. It has a predicate specifically for changes `(change ?change source target)` where the variable `change` is bound to a single change between the two AST nodes `source` and `target`. The changes can be categorized as *Insert*, *Delete*, *Move*, *Update* when a node is inserted, deleted, moved to a different location, or replaced with a different node in the AST, respectively.

QWALKEKO is language dependent and works only for Java projects. It is a plugin for the Eclipse IDE and uses the AST representation that is specific to the IDE. QWALKEKO is designed to be used by users who are familiar with the structure and concept of abstract syntax trees.

APFEL [27] is another eclipse plugin for CVS repositories that allows users to query fine-grained code changes. It works on the AST of programs and is implemented for Java. APFEL first creates token sets from the AST of the source code. These are predefined sets of tokens for every compilation unit. Some of the tokens are:

Q-*package-name*

E-*class-name*

M-*method-name*

V-*variable-name*

K-*keyword*

The tokens are identified for each version of the program and are compared against each other to determine if they were added, deleted, or updated. These tokens are then stored in a database with information such as the type of change, the corresponding version id in which the change occurred, and how often the token was changed. Users then query using SQL to retrieve information about the changes in the repository. Apart from looking for specific changes, APFEL can be used to get high-level insights about the project. For example, to query a pair of variables that are frequently inserted together, a user can issue a query that selects two variable-name tokens that were added or deleted an equal number of times.

Since the query language used in APFEL is SQL, developers who are knowledgeable in SQL can easily construct queries once they understand the different token sets tracked by APFEL. However, apart from being language and IDE dependent, it has a fixed set of tokens that are tracked. Therefore, when new language constructs are introduced, the tool needs to be updated.

Boa [7] is a source-code mining tool for large code repositories built on top of Hadoop [15]. It is implemented based on the visitor pattern which is a technique to traverse any graph or tree structure. Here, the visitor pattern is applied to the AST of a program to traverse the structure of the program. However, Boa has its own schema that represents language features and a domain-specific language to specify the queries. The query language is designed based on the concept of the visitor pattern. The AST nodes of the source code are internally mapped to the Boa schema using a language-specific mapping. The users therefore do not have to understand the structure of the source code language to query it. Currently, the Java language is fully mapped to the Boa schema.

A Boa query is compiled to a Hadoop program and run in the Hadoop framework. The advantage of using Boa over the existing programming model of Hadoop is that the domain-specific language provides a syntax that is specific to source code mining and abstracts away details like parallelism present in the Hadoop program that are not relevant in source-code mining. Boa has been evaluated against source-code mining in

large-scale and ultra-large-scale repositories and aims to enable software research using large scale data [6].

Time Warp [14] is a library of logical predicates for an existing program query language called SOUL. SOUL can be used to reason about programs in languages like Java, Smalltalk, C++, and Cobol. Time Warp adds a set of predicates that consider the history of these programs. Time Warp operates on metamodels of both the source code and its history. The metamodel for source code used here is FAMIX, which represents programs written in object-oriented languages such as Java, Python, and Smalltalk. FAMIX maps program elements such as classes, methods, and attributes and relationships between these elements similar to a UML model. The history of the source code is represented using another existing model called Hismo. Both these models use class hierarchy to represent the relationship between the object-oriented entities of the program. SOUL is a program query language implemented in Prolog that queries the structure of object-oriented programs irrespective of the language. Time Warp implements predicates similar to temporal facts in temporal logic programming that can be used to reason about the history of the software and help users navigate through the history.

Time Warp is an integrated library that consists of predicates to query the FAMIX model, Hismo model, the temporal aspect of the Hismo model, and the changes between different versions. It also includes predicates that can be used to query high-level details that span over the history of the software, such as finding all the classes that were modified between two versions, finding renamed entities, and so on. These predicates can be used to query the structure of the program and the relationship between the object-oriented entities such as classes, attributes, and methods that are modeled and tracked by the metamodels used. However, it cannot be used to reason about fine-grained code changes that do not include these entities. For example, it is not possible to query a specific change that was made in a method such as changing a `for` loop to a `for each` loop.

The Sequential Pattern Query Language (SPQL) is a query language that can be used to query sequential software engineering data. Any software engineering data that can be exhibited as a sequence of activities can be queried. SPQL has been applied to two types of sequential data: a bug report history and source code change history. A bug report history can be viewed as a sequence of status changes of the bugs raised.

A source code history is formed by a sequence of code changes. New sequential data can be plugged in by creating a schema of the changes that are tracked and an adaptor that maps domain-specific features of the new sequential data to the abstract notion of sequential data, called *snapshots*, used by SPQL. In the following, I only consider the application of SPQL to source code change history. Version metadata from a version control system such as author, date, log message, and so on is stored in a database schema and then queried using SPQL.

The queries in SPQL are similar to regular expressions matching strings except that here, regular expressions match events. Events are used to specify conditions which should hold for a snapshot. For example, the event that matches commits made on a specific date is as follows:

```
event match_date(commit_date == date )
  {commitids = add(commitids, commitId)}
```

The event compares the commit date to the specified date and if true, adds the current commit id to the list `commitids`. Here `date` and `commitids` are the SQPL variables to store intermediate results that can be used later in the query or simply returned to the user. The keyword `commit_date` is the name of the field in the database schema that stores the commit date of a version.

Following is another event in SPQL that checks if a commit message consists of the string `bug fix`.

```
event match_message(substring("bug fix", Log)) {}
```

The function `substring` is a custom function that checks for a substring match. The keyword `Log` refers to a field that stores commit message of a version.

To list all the commits that were made on a specific date and have the string `bug fix` in the commit message, the above mentioned events can be concatenated similar to regular expression concatenation.

```
match_date match_message
```

SPQL focusses on the metadata of the versions and the temporal aspect of the version history. The language does not allow users to express specific source code changes between different versions.

SCQL [16] is another language for querying software repositories using the version information. It converts the history of repositories to a graph of commits and uses temporal logic to specify queries using the version's metadata and temporal relationships between the commits.

2.3 Program Differencing

Another way to query source code changes is to identify the versions using existing tools and compare the two versions of the program using a program differencing tool. There are algorithms and tools that produce changes in different representations each catering to the specific needs of the application.

The most common program difference tool is the UNIX *diff* tool. Diff is the default differencing tool used by git command `diff` and `show` explained on page 9. Diff uses a line-based differencing algorithm that computes minimum edit distance between the source and the target file. It produces the differences as added lines, removed lines, and a combination of both along with the line number. If a line is modified, then the output will first contain the old version of the line as the deleted and the new version as an added line. This is illustrated using the following example. Consider the methods in listings 2.1 and 2.2. These are two versions of a file containing the method named `code`. The difference between the two versions is the statement that is printed to the console.

```
void code()
{
  System.out.print("This is a
     code block")
}
```

Listing 2.1: code 1

```
void code()
{
  System.out.print("This is the
     function body")
}
```

Listing 2.2: code 2

The output of the diff tool run with the above two versions is

```
- System.out.print("This is a code block")
+ System.out.print("This is the function body")
```

UNIX diff limits the ability to differentiate between line additions and deletions from line modifications. Also, if users want to look for more fine-grained changes then line-based differencing does not help.

Canfora et al. [2] came up with a line-based differencing algorithm that identifies changes as addition, deletion, or update. In this algorithm a line is added if it is not present in the previous version, deleted if it not present in the current version, and changed if it is present in both the versions but is modified. In order identify modifications to a line, the algorithm uses set-based metrics such as *Jaccard(X, Y)*, *Overlap(X, Y)* and sequence-based metrics such as *Levensthien(X, Y)* to compute the similarity between two lines X and Y.

Change Distilling by Fluri et al. [11] is a tree differencing algorithm that computes the differences between two AST nodes. This algorithm is language specific and is implemented for the Java language. The algorithm extracts fine-grained source code changes and presents them as tree operations such as *insert*, *delete*, *move*, or *update* of the AST nodes. The algorithm produces a minimum edit script consisting of the tree operations required to transform the program from one version to another. Since it computes the syntactic program difference, this algorithm has been used in tools that find refactorings in Java applications, analyze code smells, and so on.

Kodhai et al. [8] developed a rule inferencing algorithm that extracts the differences between two version of a program and represents them as change rules. Their aim is to present source-code changes to the developers as a high-level description of the change. Instead of viewing textual differences or the less obvious AST-based representation separately, the authors combine both the approaches to present a unified set of change rules.

They divide the source-code changes into the following two categories.

API level changes: This includes method renames, package renames, class renames, and other changes in the method signature. These changes are identified using textual differencing algorithms. Differentiating method renames and new methods are not possible using textual differencing. Therefore, they use a refactoring crawler tool to identify method renames.

Implementation level changes: Method-level changes such as creating new variables, changing loop conditions are identified using the LSDiff algorithm [17], which presents the structural differences such as addition and deletion of code elements and structural dependencies.

After extracting the changes in both the categories, the rule inferencing algorithm will output the changes in the form of a change rule. A change rule has three components: the scope from which the code elements are to be included, a set of code elements that are to be excluded, and transformations that describe changes between the two versions of the code elements. The elements that partially adhere to the transformations specified in the change rule are noted separately to highlight the same fact.

```
For all x : code element in (scope)
  except(exceptions)
  transformation(x)
```

The set of transformations is fixed and predefined. Some of the API level transformations are `packageRename(..)`, `classRename(..)`, `argReplace(..)`, which contain corresponding arguments that bind to the value before and after. Implementation-level transformations are only addition and deletion of code elements and include `field(..)`, `calls(..)`, and so on For example, the following rule states that all the `draw` methods in the class `Shape` defined within the package `shapes` were renamed to `render`. The wild card character `*` is used to denote any value.

```
For all x: shapes.*.draw(*)
  methodRename(x,draw,render)
```

Chapter 3: Background

GitQL queries operate on the choice calculus representation of software repositories. This chapter explains the choice calculus and the choice edit model that is used as a basis for how repositories can be stored in the choice calculus representation. Further, regular expressions for specifying patterns are explained. GitQL extends regular expressions by choice patterns.

3.1 The Choice Calculus

A software repository can be viewed as a sequence of changes made to files. A variant corresponds to a version of a file. When committing a change, a user is creating a new variant of the file. The latest variant of the file is the result of all the previous operations on the initial version of the file. An intermediate variant consists of all changes made up to the selected variant but does not contain any of the later changes made to it. Using this view of software repositories, we can represent a version-controlled software project in the choice calculus notation.

The choice calculus [10] is a formal language to represent variations. It is a generic language that can be applied to an arbitrary object language. The choice calculus employs *choices* to denote variations. The syntax of a choice is $D\langle A, B, \dots \rangle$ where A, B, and so on are called alternatives. The alternatives could be terms in the object language or choices themselves. Each of the alternatives in a choice leads to a different variant. Therefore, a choice represents all possible variations of an expression in the object language from which one can be selected.

In the case of software repositories, the object language is string. A choice represents the textual difference between two versions of a file. Therefore, the number of alternatives in a choice is limited to two resulting in what is called a binary choice. The first alternative consists of the text before the modification and the second alternative consists of the text after the modification. The first and the second alternatives of a binary choice are also referred to as the left and the right alternatives respectively.

Consider the example of two different implementations of a function `twice`.

```
int twice(int x) { return x+x; }

int twice(int x) { return 2*x; }
```

We can capture the variation between the two implementations of the function `twice` in the choice expression.

```
int twice(int x) { return A⟨x+x, 2*x⟩; }
```

Here, both the alternatives of the choice are terms in the object language, in this case, C. A choice is always bound to some dimension D. Two choices can have same or different dimension names. To understand this better, consider another implementation where the variable is named as `y`. Therefore, we now have two additional implementations of the function `twice`.

```
int twice(int y) { return y+y; }

int twice(int y) { return 2*y; }
```

The choice expression now has two dimensions, representing a variation in the variable name and function body respectively.

```
int twice(int B⟨x, y⟩) { return A⟨B⟨x, y⟩+B⟨x, y⟩, 2*B⟨x, y⟩⟩; }
```

Two choices with dimension name B are nested in the choice with dimension name A thereby representing variations on multiple levels. This way, all the variants of a program in the object language can be represented using the choice calculus in one variational program.

A plain program variant can be obtained from a variational program consisting of choices by recursively selecting an alternative from each choice until all the choices are eliminated. The process of selecting an alternative from a choice is called *selection*, which uses selectors. A selector is a mapping from a dimension name to an alternative. For binary choices we have the two selectors $D.l$ and $D.r$ where D is the dimension name and l, r are abbreviations for left and right. $D.l$ selects the left alternative of a choice C, and $D.r$ selects the right alternative. The set of selectors required to obtain a concrete program is called a *decision*.

To obtain the variant of `twice` which has `y` as the variable and is implemented using `+`, we have to make two selections; one for the choice of variable names and the other for

Edit operation	Text before	Text after	Choice created
Insert	`""`	`Y`	$D\langle$, `Y`$\rangle$
Delete	`X`	`""`	$D\langle$`X`, $\rangle$
Update	`X`	`Y`	$D\langle$`X`, `Y`$\rangle$

Table 3.1: Choices for basic edit operations

the choice of the function body. The decision $\{A.l, B.r\}$ will select the left alternative from the choice with dimension name A and the right alternative from the choices with dimension name B and produce the following plain program in the object language.

```
int twice(int y) { return y+y; }
```

Here $B.r$ selects the second alternative from all the three choices that have the dimension name B. The dimension names, therefore, synchronize the changes that are made in different places but are all part of the same variation.

3.2 The Choice Edit Model

The choice edit model [9] is used to represent the version history of a file in the choice calculus representation. The choice edit model is a program edit model that can be used to understand and reason about the edits that a developer makes while editing a program. It defines how textual edits to a file can be encoded as choices. When a part of a program P is changed to Q, a choice between P and Q is introduced. Here, the left alternative consists of the old value and the right alternative consists of the new value. Edits are obtained from the three basic operations: insert, delete, and update. The choices introduced for each of the basic edit operations follow the pattern shown in table 3.1.

Consider a C program `P` which consists of a function `f`. This is the first version of the function.

```
P = int f(int a)
    { int b; return a+b; }
```

A programmer makes the following sequence of edits on `f`. After each edit, choices are created in the variational program and the decision corresponding to new version of

the program is updated.

Edit 1: Change the function argument to `c`. In the variational program `VP1`, a choice with dimension name A between `a` and `c` is introduced in all the places where the edit is made. The decision `D1` that corresponds to the newly created version has a selector $A.r$ that selects the right alternative of choice A consisting of the changes made in this edit in `VP1`.

```
P1 = int f(int c)
     { int b; return c+b; }

VP1 = int f(int A⟨a,c⟩)
      { int b; return A⟨a,c⟩+b; }

D1 = {A.r}
```

Edit 2: Assign the value `1` to the variable `b`. A choice with dimension B between an empty string and "`=1`" is introduced. This edit is an insert operation and therefore has no old value. The decision for this version of the program will be `D1` $\cup \{B.r\}$ where $B.r$ is the selector for the choice B selecting all the changes introduced in this edit. The decision `D1` from the previous version is propagated to `D2` so that when `D2` is applied to the variational program `V2`, all the edits that resulted in `P1` are seen in `P2` as well.

```
P2 = int f(int c)
     { int b=1; return c+b; }

VP2 = int f(int A⟨a,c⟩)
      { int bB⟨,=1⟩; return A⟨a,c⟩+b; }

D2 = {A.r, B.r}
```

Edit 3: Undo the assignment of variable `b` that was made in the previous version. No new choices are created for this edit. Instead, the decision is updated to select the left alternative of B that consists of the text before edit 2 was made, which is an empty string. After this edit, the variational program `VP3` will be the same as `VP2` since no new choices have been added.

```
P3 = int f(int c)
     { int b; return c+b; }
```

```
VP3 = int f(int A⟨a,c⟩)
      { int bB⟨,=1⟩; return A⟨a,c⟩+b; }

D3 = {A.r, B.l}
```

Edit 4: Rename the function argument again, now to `d`. A new choice with the dimension C is introduced. This edit changes the string that has already been changed previously, and therefore choice C is nested in the right alternative of choice A. Such edits are called chain edits. The resulting decision is $\texttt{D4} = \texttt{D3} \cup \{C.r\}$.

```
P4 = int f(int d)
     { int b; return d+b; }

VP4 = int f(int A⟨a,C⟨c,d⟩⟩)
      { int bB⟨,=1⟩; return A⟨a,C⟨c,d⟩⟩+b; }

D4 = {A.r, B.l, C.r}
```

Edit 5: Assign `2` to the variable `b`. A new choice with dimension name D between an empty string and "`=2`" is created. The new choice is nested in the left alternative of choice B because the previous version does not have edit 2 (denoted by the selector $B.l$ in `D4`) as a result of the undo operation in edit 3. Now, there are two different edits made at the same location to the same string, in this case an empty string. This scenario is called a branch edit.

```
P5 = int f(int d)
     { int b=2; return d+b; }

VP5 = int f(int A⟨a,C⟨c,d⟩⟩)
      { int bB⟨D⟨,=2⟩,=1⟩; return A⟨a,C⟨c,d⟩⟩+b; }

D5 = {A.r, B.l, C.r, D.r}
```

Multiple edit operations that involve changes in different locations in a file can be synchronized by having the same dimension name for the corresponding choices. Therefore when selections are made, all the choices that have the same dimension name will have either the left or the right alternative selected. The variational program `VP5` now

consists of the information on how the program was edited. It consists of all the edit operations that were made to the initial program `P`.

All the program variants can be obtained from the variational program by applying decisions that correspond to each of the variants. For example, the decision $\{A.r, B.l, C.r, D.l\}$ applied to `VP5` will result in `P4`.

```
int f(int d)
{ int b; return d+b; }
```

In the choice edit model, a programmer can not only view the program variants created by her but also some of the new variants that were not visible during the editing process. For example, the decision `D6` $= \{A.l, B.l, D.r\}$ applied to `VP5` will result in a program variant that only consists of edits 3 and 5.

```
int f(int a)
{ int b=2; return a+b; }
```

Note that `D6` does not contain a decision for the choice C. This is because choice C is nested in the right alternative of choice A and is not present as an independent choice elsewhere in `VP5`. The selector $A.l$ eliminates the need for a selector for choice C altogether. All the nested choices are fully dependent on the choice they are nested in and partially dependent in case they exist as independent choices elsewhere in the variational program.

Version-controlling a file is the same as program editing. A commit in git corresponds to an edit in the choice edit model. A commit consolidates multiple small edits into one large edit. For each commit that introduces changes in a file, choices can be added like in the choice edit model. All the choices corresponding to a commit will have the same dimension name. This way we can identify all the changes made by a specific commit. Using the choice edit model, we can create a variational file that contains the entire version history of the file represented using the choice calculus.

3.3 Regular Expressions

A regular expression is a string for describing a search pattern. Using this pattern, a regular expression engine looks for all strings that match the pattern. The basic definition of a regular expression language consists of a set of character literals, wildcards,

and operators that combine regular expressions. Some of the operations used in GitQL patterns are:

1 Concatenation: A regular expression can be formed by concatenating two or more regular expressions. For example, let `a` and `b` be two regular expressions that match characters 'a' and 'b', respectively, then `ab` matches the string "ab". Similarly, concatenation of the regular expressions `ab` and `c` will match the string "abc".

2 Alternation: An alternation pattern `P|R` matches strings that conform to either of the regular expressions P and R. For example, `ab|xy` matches both the strings "ab" and "xy".

3 Any Character: The dot symbol (.) is used to match any character. In GitQL, we use a similar operation called *any string* denoted by the underscore symbol (_) that matches any string. For example, `_xy` matches all the strings that end with "xy" such as "xy", "axy", "abxyxy", and so on.

Chapter 4: GitQL

GitQL is a query language for software repositories that enable the users of a software repository to query textual changes made to the source code. It is based on the choice calculus representation of software repositories. GitQL queries are are based on vgrep, which is a generalization of grep for variational strings. At the heart of a GitQL query is the *match* statement. A match statement has the following structure:

var ← `match` *pattern* `in` *source* `where` *conditions*

A match statement consists of a pattern, a source, and optional conditions to filter the result. Vgrep will not just look for the pattern in a single version of the source file but through all of the versions. Patterns in GitQL extend a basic regular expression language. The *where* conditions can be used to filter match results based on the commit metadata or by comparing match results. Explanations for each of the parameters follows in the subsequent sections.

The following section describes the syntax of GitQL.

4.1 GitQL Syntax

query ::= *values* `from` *search*

search ::= *var* ← `match` *pattern* `in` *source* `where` *conditions*
| *search*`,`*search*

var ::= *string*

values ::= *value* | *value*`,`*values*

value ::= *var*
| `pos` *var*
| `count` *var*

source ::= `-f` *filename*
| *var*
| `{` *query* `}`

pattern ::= `"`*string*`"`
| _
| *pattern pattern*
| *pattern*`|`*pattern*
| *commit*`<`*pattern*`,`*pattern*`>`
| *var*
| *none*

commit ::= `#`*string* | *var*

conditions ::= *condition*
| *condition bool-op conditions*
| `(`*conditions*`)`

bool-op ::= `and` | `or` | `not`

condition ::= *commit-info-comp*
| *result-comp*

result-comp ::= *var rel-op var*

commit-info-comp ::= *commit-info rel-op commit-info*

rel-op ::= `==` | `>` | `>=` | `<` | `<=` | `/=`

commit-info ::= *commit*`.date`
| *commit*`.author`
| `"`*yyyy*`/`*mm*`/`*dd*`"`
| *author-name*

author-name := string

In the following I will explain the main components of a GitQL query in more detail

4.1.1 Source

A source can be a filename, a variable, or another GitQL query. The source file is the file whose history needs to be queried. Although, vgrep works on variational files, users specify just the original source file name in the query. Since variational files are present in the same location as the corresponding source files, mapping from the original source file to its variational file is done internally by the search engine relieving the users from knowing the naming conventions of variational files. Variables and nested queries are explained later.

4.1.2 Pattern

A string in a pattern is a sequence of ASCII characters. Certain characters have predefined meaning in GitQL and therefore need to be escaped using a backslash. The underscore (_) is used to specify any string.

A sequence of patterns can be formed by concatenating patterns. This way, users could search for a string `ab` or a string followed by a choice pattern (see below) `int` $1\langle$`foo`$,$`bar`$\rangle$ and so on.

An alternation pattern can be used to match a single pattern out of multiple possible patterns. Each pattern is matched starting from left and returns the first successful match. For example, the pattern `foo | bar` can be used to find if there is `foo` or `bar` in the file. Vgrep will first try to match `foo` and if unsuccessful, it will try to match `bar`.

A choice pattern $C\langle$`P1`$,$`P2`$\rangle$ is used to specify committed changes. It consists of two sub-patterns `P1` and `P2` in the left and the right alternative of the choice pattern, respectively. The pattern `P1` is used to match the text before the change, and the pattern `P2` is used to match the text after the change. Based on the value of C, a choice pattern can be used in the following two ways:

1. Look for a change that occurred in a specific commit by specifying the commit id. Vgrep then tries to match the patterns only against the changes made in the specified commit. For example, with the pattern `#1`$\langle$`foo`$,$`bar`$\rangle$, vgrep tries to match the change from `foo` to `bar` only if it is in commit 1.

2. Look for a specific change in the entire version history of the file and obtain the commit id. In this case, a variable is used that will be bound to the commit id that is

matched. Vgrep will try to match both sub-patterns and if successful, the commit id is bound to the commit variable in the choice pattern. For example, the pattern $d\langle$`foo`, `bar`$\rangle$ matches the change from `foo` to `bar` made in any of the commits. For each match, the corresponding commit id is bound to the commit variable d. Commit variables are useful when the commit id is unknown or when a change that the user is trying to find could have been made in many different commits. Each match returned by the query will have a commit id to which the commit variable is bound.

Choice patterns and alternation patterns are similar in that both can be used to match multiple patterns. However, using alternation patterns one can only look for matches within a single version of a file whereas choice patterns are used to search through the differences between multiple versions of a file.

Variables in a pattern can be used to refer to a part of the query result. This is specifically useful for the sub-patterns in a choice pattern when the user just knows the pattern either for the text before the change or for the text after the change. A variable can be in the left alternative, the right alternative, or in both the alternatives. Let `P`, `Q` be the sub-patterns of a choice pattern with a commit variable d and `q`, `r` be the variables. The usage of variables in a choice pattern is shown in the following.

$d\langle$`P`, `q`$\rangle$: When a variable is in the right alternative, vgrep will find all the changes that matches the text before the change using `P` and then binds the text after the change to the variable `q`. If there are further changes to the same text, then all those changes will also be bound to the query variable. For example, in the pattern $d\langle$`foo`, `q`$\rangle$, the query variable `q` is bound to all the changes made to the text `foo`.

$d\langle$`q`, `P`$\rangle$: When a variable is in the left alternative, vgrep will find all the changes in which the new text matches `P` and the old text that was changed will be bound to `q`.

$d\langle$`q`, `r`$\rangle$: When query variables are in both the alternatives, vgrep will match every change made to a file. The query variables `q` and `r` will be bound to the old text and the new text of every change, respectively.

Variables in a choice pattern can also be concatenated with other patterns. For example, with the choice pattern $d\langle$`"ab"q`, `"cd"r`$\rangle$, vgrep matches all the changes in which the text beginning with `ab` was changed to the text beginning with `cd`. The

variables `q` and `r` are bound to the remainder of the texts before and after the change, respectively. Variables can also be used without choice patterns. For example, the pattern `"ab"x"lm"` matches the string that begins with `ab` followed by any plain string or variational string and ends with `lm`. The variable `x` is bound to the substring of the match that is in between `ab` and `lm`.

4.1.3 Where Conditions

A match statement consists of an optional field that can be used to filter specific matches from the result. Each condition is a predicate consisting of relational operators that operates either on commit metadata such as author name and commit date or the match result itself. Multiple conditions can be combined using the boolean functions `and`, `or`, and `not`.

4.1.4 Variables and Nested Queries

Variables in GitQL are used in two ways. They can be used in a pattern as explained in section 4.1.2. The other use of variables is to refer to the match results that are obtained after executing the match statements. The reason for having variables is that the match results can be reused by other match statements as the source in place of a source file. The match statements that have variables as the source search within the match results which, depending on the pattern, are specific patches of version history rather than the entire history of the source file. Therefore, users can break down a complex query into multiple match statements by using match results from previously defined match statements. Also, if there are multiple match statements, users can selectively choose to display only some of the results by picking the corresponding variables.

A match statement can have a GitQL query as the source in which case the nested query result will be used for further matching. Variables used in a nested query are not accessible to the outer queries.

4.2 Examples

Consider the following short history of a Java program `Employee.java`. Listing 4.1 shows the initial version of the program and listings 4.2 - 4.13 show the changes made in each commit. Version 4.2 is created in a different branch. Version 4.4 is created after a merge between version 4.2 and version 4.3. Figure 4.1 shows the DAG representation of the changes.

```
import java.util.LinkedList;

public class Employee{

 private String name;
 private LinkedList<Designation> designations;

 public void setName(String name)
 {
    this.name = name;
 }

 public String getName()
 {
    return this.name;
 }

 public void setDesignation(Designation designation)
 {
    this.designations.push(designation);
 }

 public Designation getdesignation()
 {
    return this.designations.getFirst();
 }
}
```

Listing 4.1: Version 1 of a Java program

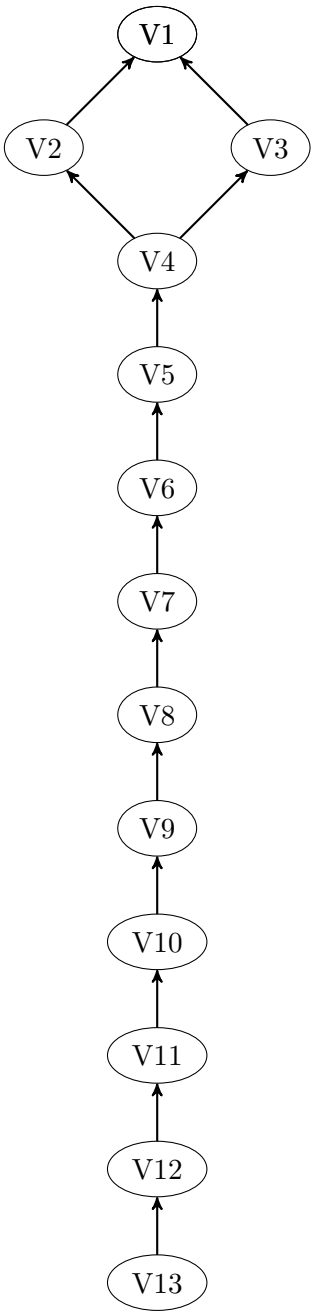


Figure 4.1: A DAG of the commit history of `Employee.java`

```
...
- public Designation getdesignation()
+ public Designation getLatestDesignation()
  {
    return this.designations.getFirst();
...
```

Listing 4.2: Version 2 - Renamed `getdesignation()` in another branch

```
...
- public Designation getdesignation()
+ public Designation getFirstDesignation()
  {
    return this.designations.getFirst();
  }

+ public LinkedList<Designation> getAllDesignations()
+ {
+    return this.designations;
+ }
+
```

Listing 4.3: Version 3 - Renamed `getdesignation()` and added a new method

```
...
- public Designation getFirstdesignation()
 - public Designation getLatestdesignation()
++ public Designation getCurrentDesignation()
  {
    return this.designations.getFirst();
...
```

Listing 4.4: Version 4 - Merged version 2 and 3 and the resolved merge conflict

```
...
 private LinkedList<Designation> designations;
+ private BigDecimal salary;

...
```

```
+ public BigDecimal getSalary()
+ {
+    return this.salary;
+ }
+
+ public setSalary(float percent)
+ {
+    Designation curr = this.getCurrentDesignation();
+    this.salary = curr.getBaseSalary().multiply(new BigDecimal(1+ percent/100));

+ }
+
}
```

Listing 4.5: Version 5 - Added a new member variable and getter-setter methods

```
...
- public setSalary(float percent)
+ public void setSalary(float percent, BigDecimal extra)
  {
     Designation curr = this.getCurrentDesignation();
-    this.salary = curr.getBaseSalary().multiply(new BigDecimal(1+ percent/100));

+    this.salary = curr.getBaseSalary().multiply(new BigDecimal(1+ percent/100)).add(extra);

  }
...
```

Listing 4.6: Version 6 - Updated the `setSalary` method

```
 ...
     this.salary = curr.getBaseSalary().multiply(new BigDecimal(1+
        percent/100)).add(extra);
+    //Ensure that the salary doesnot go beyond the maximum
                            allowed for the specific designation
+    if(this.salary.compareTo(curr.getMaxSalary()) == 1)
+    {
```

```
+      /* Don't provide the extra component
+      TODO: confirm*/
+      this.salary = this.salary.subtract(extra);
+    }
  }
 ...
```

Listing 4.7: Version 7 - Added maximum-salary-check in the `setSalary` method

```
...
     /* Don't provide the extra component
     TODO: confirm*/
-     this.salary = this.salary.subtract(extra);
+     this.salary = curr.getMaxSalary();
 }
...
```

Listing 4.8: Version 8 - Update the `setSalary` method

```
...
- public void setSalary(float percent, BigDecimal extra)
+ public void setSalary(float percent, BigDecimal extra) throws UpdateException
 {
    Designation curr = this.getCurrentDesignation();
    this.salary = curr.getBaseSalary().multiply(new BigDecimal(1+
       percent/100)).add(extra);
    //Ensure that the salary doesnot go beyond the maximum allowed
        for the specific designation
    if(this.salary.compareTo(curr.getMaxSalary()) == 1)
-    {
-      /* Don't provide the extra component
-      TODO: confirm*/
-      this.salary = curr.getMaxSalary();
-    }
+      throw new UpdateException("Current salary "+ this.salary +
" is greater than the max salary"); this.salary = BigDecimal.Zero;
 }
...
```

Listing 4.9: Version 9 - Throw an exception

```
...
  private BigDecimal salary;
+ private Integer age;

+ public Integer getAge()
+ {
+    return age;
+ }
+
+ public void setAge(Integer age)
+ {
+    this.age = age;
+ }
+
...
```

Listing 4.10: Version 10 - Added a new variable and getter-setter methods

```
...
public void setSalary(float percent, BigDecimal extra) throws
    UpdateException
{
-    Designation curr = this.getCurrentDesignation();
-    this.salary = curr.getBaseSalary().multiply(new BigDecimal(1+ percent/100)).
     add(extra);
+    Designation currDesig = this.getCurrentDesignation();
+    this.salary = currDesig.getBaseSalary().multiply(new BigDecimal(1+ percent/100)).
     add(extra);
     //Ensure that the salary doesnot go beyond the maximum allowed
         for the specific designation
-    if(this.salary.compareTo(curr.getMaxSalary()) == 1)
+    if(this.salary.compareTo(currDesig.getMaxSalary()) == 1)
       throw new UpdateException("Current salary "+ this.salary + "
           is greater than the max salary"); this.salary =
          BigDecimal.Zero;
}
...
```

Listing 4.11: Version 11 - Renamed variable `curr` to `currDesig`

```
...
- public LinkedList<Designation> getAllDesignations()
+ public LinkedList<Designation> getDesignationHistory()
...
```

Listing 4.12: Version 12 - Renamed the `getAllDesignation` method to `getDesignationHistory`

```
...
+ public LinkedList<Designation> getDesignationHistory()
- {
-    return this.designations;
- }
-
...
```

Listing 4.13: Version 13 - Deleted the `getDesignationHistory` method

4.2.1 Queries and Results

Based on the above listed history of the Java program, here are a few queries and the results obtained after its the execution by vgrep. The results are shown in the choice calculus notation. The output has been modified to show the version numbers that mark each of the version in the given example instead of the commit hash generated by git. This is to easily map the version numbers in the example.

Query 1: Match all the occurrences of `public`.

```
ms from ms ← match "public" in -f Somefile.java
```

Result:

```
ms:
Match 1: public
Match 2: 5<,10<,public>>
Match 3: 5<,10<,public>>
Match 4: public
Match 5: public
```

Match 6 : `public`
Match 7 : `public`
Match 8 : `3<,13<public,>>`
Match 9 : `5<,public>`
Match 10 : `5<,public>`

Query 2: Find the commit that renamed the variable `curr` to `currDesig`.(Show only the dimension bindings for the variable name `d`.)

```
d from _ ← match d<"curr","currDesig"> in -f Employee.java
```

Result:

```
d:[11]
```

Query 3: Show all the changes made in version 11.

```
m from m ← match #11<_,_> in -f Employee.java
```

Result:

```
m:
Match 1 : 5<,11<curr,currDesig>>
Match 2 : 5<,11<curr.getBaseSalary().multiply(new,currDesig.
    getBaseSalary().multiply(new>>
Match 3 : 5<,7<,11<if(this.salary.compareTo(curr.getMaxSalary()),if(
    this.salary.compareTo(currDesig.getMaxSalary()))>>>
```

Query 4: Show the commit and the variable binding for the changes that were made to the method `getAllDesignations`.

```
x, d from _ ← match d<"getAllDesignations()",x> in -f Employee.
    java
```

Result:

```
x:
Match 1 : 13<getDesignationHistory(),>

d:[12]
```

Query 5: Show all the changes made previously to the function `getCurrentDesignation`.

```
x from _ ← match d<x,"getCurrentDesignation"_>
  in -f Employee.java
```

Result:

```
x:
Match 1: getFirstDesignation()
Match 2: 2<getdesignation(),getLatestDesignation()>
```

Query 6: Show the position of all the matches from query 5.

```
pos x from _ ← match d<x,"getCurrentDesignation"_>
      in -f Employee.java
```

Result: The positions shown in the result are with respect to the variational string. These positions can be used to show the matches in a user interface. The variational positions in itself do not offer any insight to the users currently.

```
pos x:P 3 (Right (NoPos,P 0 (Right (P 0 (Left 0),NoPos))))
      P 3 (Right (P 0 (Right (P 0 (Left 0),P 0 (Left 0))),NoPos))
```

Query 7: Show all the insert edits made to the file.

```
m from m ← match d<,x> in -f Employee.java
```

Result:

```
m:
Match 1: 5<, private BigDecimal salary;
>
Match 2: 5<,10<, private Integer age;

 public Integer getAge()
 {
    return age;
 }

 public void setAge(Integer age)
```

```
 {
    this.age = age;
 }
>>
Match 3: 3<,
 }>
Match 4: 5<,6<,9<, throws UpdateException>>>
Match 5: 5<,7<, //Ensure that the salary does not go beyond the
   maximum allowed for the specific designation
    >>
```

Query 8: Show the commit when `extra` was added in the `setSalary` method.

```
m from _ ← match d<,_"extra"_> in -f Employee.java
```

Result:

```
m:
Match 1: 3<,
 }>13<
 },>5<,6<percent),percent, BigDecimal extra)>>
Match 2: 3<,
 }>13<
 },>5<,6<percent/100));,percent/100)).add(extra);>>
```

Query 9: Show all the changes made between 11-15-2017 and 11-17-2010.

```
m from m ← match d<x,y> in -f Employee.java
  where d.date >= "2017/11/15" and d.date <= "2017/11/17"
```

Result:

```
m:
Match 1: 5<, private BigDecimal salary;
>
Match 2: 5<,10<, private Integer age;

 public Integer getAge()
 {
```

```
    return age;
 }

 public void setAge(Integer age)
 {
    this.age = age;
 }
>>
```

Query 10: Count the number of matches from query 5.

```
count x from _ ← match d<x,"getCurrentDesignation"_>
        in -f Employee.java
```

Result:

```
count x: 2
```

Chapter 5: Implementation

In this chapter I discuss the implementation details of three main components of the thesis: the choice encoding of software repositories, the design rationale of types involved in GitQL, and vgrep.

5.1 Choice Encoding

GitQL is designed to query the history of a software repository stored as a variational string in the choice calculus representation. Hence the first step is to translate the text files in a git repository to the choice calculus notation. Creation of variational strings involves encoding changes as choices, hence, the name choice encoding. Variational strings are created for each text file and stored in a separate file in the same location as the source file with a `.v` extension. These files are called variational files. For example, the variational string of the file README.md will be stored in the variational file README.md.v. The variational file of a source file contains its entire version history encoded as choices. The choice encoding process is tightly coupled with git because of the built-in commands used to interact with git and it can only be used to encode git repositories.

The core of the algorithm is to identify the changes made in git commits that correspond to chain edits and branch edits of the choice edit model explained in section 3.2 of chapter 3. The choice encoding tool called as GitEncode is built on top of the CCTrack function from Wyatt Allen's thesis on variational parsers [1].

The following steps are involved in the choice encoding process of a git repository.

1. Create a directed acyclic graph (DAG) of the repository from its commit history. The root of the DAG is the initial commit in the branch the user wants to query the history of. Ideally this should be the branch that acts like a main branch from which all the other branches (which could be further branched) are forked and merged back. Commits from all such branches are considered in the choice encoding process. The DAG representation will allow us to see all the commits that

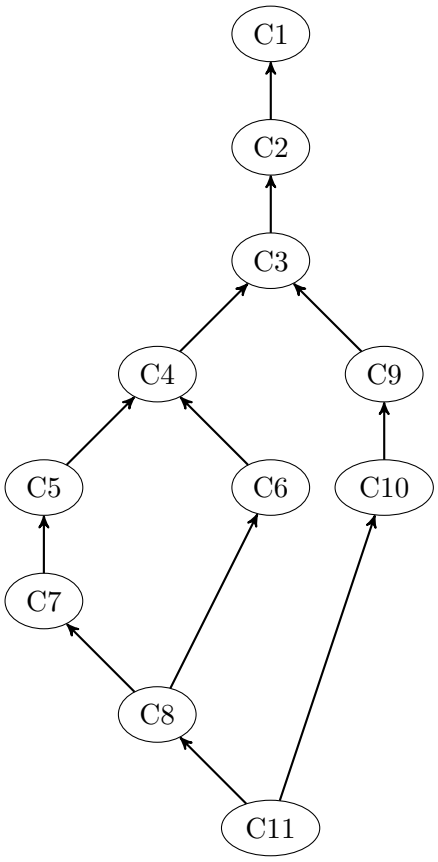
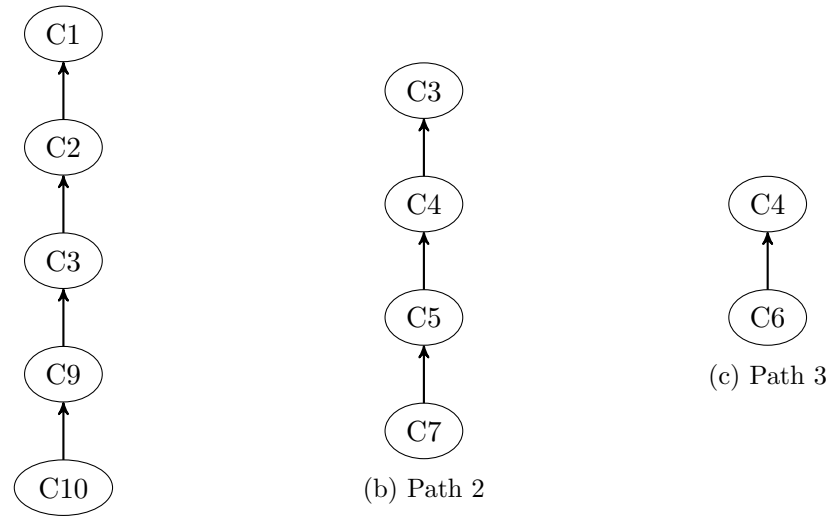


Figure 5.1: A DAG of the commit history

were made in different branches and hence encode the changes in these commits as per the choice edit model. Figure 5.1 shows an example DAG of the version history of a file.

2. Identify the difference between two versions of a file starting from the root commit. If the file was added in the root commit, the contents of the file are copied as is to a new variational file, since there are no variations in it yet. All the commits from then on make some textual modifications to the file. For each commit, the changes introduced are identified by performing a difference operation between the version of the file created by the current commit and the version of the file created by its parent commit. The differences obtained will determine the type of the edit



(a) Path 1 (b) Path 2 (c) Path 3

Figure 5.2: Encoding linear edits

operation (insert, delete, or update). The changes are then annotated with choices in the same location where the changes were made and hence the positions of these changes do not have to be maintained. Also, the choices introduced for a commit will have the same dimension name. The mappings between dimension names and the corresponding git commit ids are maintained externally in a separate file which is called the *metafile* and is used by vgrep. Metafiles files have a .m extension. The new choices are then merged into the previous variational string using the CCTrack function [1]. All the commits that have a single parent are encoded this way. Figure 5.2 shows the different paths of commits that would be encoded. These paths only include commits that have a single parent. The nodes in these paths can be collapsed into a single node that consists of all the changes applied sequentially. The resulting DAG is shown in 5.3. Node C123 consists of all the changes of commits C1, C2, and C3. Similarly, C57 and C910 consists of all the changes of commits C5 and C7 and of commits C9 and C10, respectively.

3. Merge the variational strings. Merge commits have to be handled differently, since the changes are being taken from two or more commits and they could potentially have conflicts. In git, two versions of a file that have changes in the same line will cause

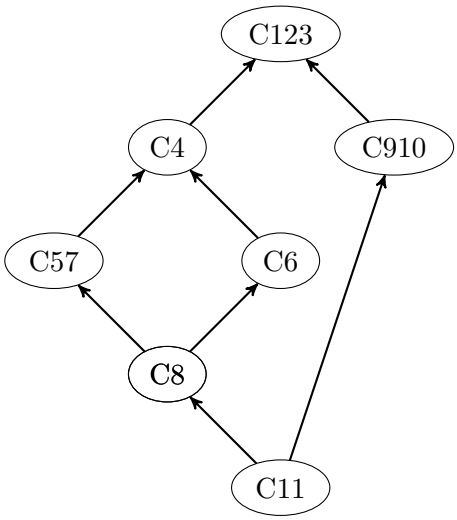


Figure 5.3: Compressed DAG

a conflict. However, users can set the changes from one branch to be preferred over the other using the git merge options `-X ours` or `-X theirs`. Since there are no reliable ways to identify if there was a conflict from the commit metadata, a git merge between the parents is performed once again. The merge operation is aborted after identifying the conflicted files, and the repository is returned back to its original state. In git merge, the branch in which the user is currently on is referred to as the *first parent* and the other parent is the branch that is being merged in. When variational strings of two parents are merged, the first parent is given preference for nesting the choices. This is explained in the following.

When a merge commit is encountered, the variational strings of two of its parents are merged (and then recursively merged with the remaining parents). These are the rules for merging non-conflicting variational strings of two parents: Let

- P_1 and P_2 be the parent commits of a merge commit; P_1 being the first parent,
- μ be a binary operator that merges two variational strings, in this case $\mu(P_1, P_2)$,
- X, Y and X', Y' be parts of the variational strings of P_1 and P_2 respectively,
- L, L' be the left alternatives and R, R' be the right alternatives of two choices

respectively,

- V_k be the view decision for a variational string K,
- ν be a binary operator that merges two view decisions, and
- M be the merged variational string.

Rule 1: When a plain string S is unchanged in both the parents, S is left unchanged in M.

$$\mu(XSY, X'SY') \rightarrow \mu(X, X')S\mu(Y, Y')$$
$$V_m \rightarrow \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'})$$

Rule 2: When a plain string S is changed in either of the parents and thereby introduces a new choice C, then the choice is copied to M, and S is discarded. The view decision is updated to reflect the selector for the new choice.

$$\mu(XSY, X'C\langle S, R\rangle Y') \rightarrow \mu(X, X')C\langle S, R\rangle\mu(Y, Y')$$
$$\mu(XC\langle S, R\rangle Y, X'SY') \rightarrow \mu(X, X')C\langle S, R\rangle\mu(Y, Y')$$
$$V_m \rightarrow \{C.r\} \cup \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'})$$

Rule 3: When an existing choice is updated by either of the parents, then in M the alternatives of the choice are merged.

$$\mu(XC\langle L, R\rangle Y, X'C\langle L', R'\rangle Y') \rightarrow \mu(X, X')C\langle\mu(L, L'), \mu(R, R')\rangle\mu(Y, Y')$$
$$V_m \rightarrow \nu(V_l, V_{l'}) \cup \nu(V_r, V_{r'}) \cup \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'}) \cup V_{cp}$$
where V_{cp} is the view decision that consists of the selection for choice C present in both the parents.

When a merge results in conflicts, the conflicting parts of the parents' variational strings are merged using the following rule.

Rule 4: Two different choices are encountered in P_1 and P_2 when there are two different edits made at the same location in each of the branches being merged. This corresponds to branch edits in the choice edit model. The conflicting choice of P_2 will be nested in the left alternative of the conflicting choice of P_1, since P_1 is the first parent. The rule has four different cases based on how the conflicts are resolved in git:

Case 1: When the changes of P_2 are overridden by the changes of P_1 (using the `-X ours` option), the conflicting changes from P_1 are selected.

Case 2: When the changes of P_1 are overridden by the changes of P_2 (using the `-X theirs` option), the conflicting changes from P_2 are selected

In both these cases the merged variational string will be the same.
$\mu(XC\langle L, R\rangle Y, X'C'\langle L', R'\rangle Y') \rightarrow \mu(X, X')C\langle C'\langle \mu(L, L'), R'\rangle, R\rangle \mu(Y, Y')$

The view decision, however, will change based on which parent's changes are selected.
Case 1: $V_m \rightarrow \{C.r\} \cup V_r \cup \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'})$

Case 2: $V_m \rightarrow \{C.l, C'.r\} \cup V_{r'} \cup \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'})$

Case 3: When both the changes are included as part of the conflict resolution, then the choices are simply concatenated in the order they appear in the merged file, and both the changes are selected.
$\mu(XC\langle L, R\rangle Y, X'C'\langle L', R'\rangle Y') \rightarrow \mu(X, X')C\langle L, R\rangle C'\langle L', R'\rangle \mu(Y, Y')$
or
$\mu(XC\langle L, R\rangle Y, X'C'\langle L', R'\rangle Y') \rightarrow \mu(X, X')C'\langle L', R'\rangle C\langle L, R\rangle \mu(Y, Y')$

$V_m \rightarrow \{C.r, C'.r\} \cup V_{r'} \cup \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'})$

Case 4: When the conflict is resolved by making a different change altogether, the merged variational string is compared with the new change.
Step 1: $\mu(XC\langle L, R\rangle Y, X'C'\langle L', R'\rangle Y') \rightarrow \mu(X, X')C\langle C'\langle \mu(L, L'), R'\rangle, R\rangle \mu(Y, Y')$
Step 2: $\mu(XC\langle C'\langle \mu(L, L'), R'\rangle, R\rangle Y, X'D\langle L'', S\rangle Y') \rightarrow$
$\mu(X, X')C\langle C'\langle \mu(L, L'), R'\rangle, D\langle R, S\rangle\rangle \mu(Y, Y')$
$V_m \rightarrow \{C.r, D.r\} \cup \nu(V_x, V_{x'}) \cup \nu(V_y, V_{y'})$

```
...
1⟨contains, 2⟨elem, present⟩⟩ :: Eq a => a -> Tree a -> Bool
1⟨contains, 2⟨elem, present⟩⟩ x Leaf = False
1⟨contains, 2⟨elem, present⟩⟩ x (Node y l r) = x == y ||
    1⟨contains, 2⟨elem, present⟩⟩ x l ||
    1⟨contains, 2⟨elem, present⟩⟩ x r
```

Listing 5.1: Encoding of version history of a program file

```
type SearchQuery = [MatchGen]

data MatchGen = MatchGen Var Pattern Source (Maybe Conditions)
```

Listing 5.2: The type of a match generator

A section of a variational file is shown in the listing 5.1. It shows a Haskell function initially named as `contains` is renamed to `elem`, and then again to `present`.

5.2 Abstract Syntax

GitQL is a deeply embedded domain-specific language implemented in Haskell. The abstract syntax is derived from the concrete syntax listed in chapter 4. Every match statement in a GitQL query corresponds to a match generator represented by the type `MatchGen`, and a GitQL query is a list of all the match generators. Listing 5.2 shows the high-level types involved.

The type of a pattern used in GitQL is shown in listing 5.3.

```
data Pattern = Plain Char
             | Seq Pattern Pattern
             | Alt Pattern Pattern
             | PChc DimTy Pattern Pattern
             | Any
             | None
             | QVar Var
```

Listing 5.3: The type of a pattern

```
data VString = Str String | Chc Dim VString VString
```

Listing 5.4: The type of a variational string as a choice-tree

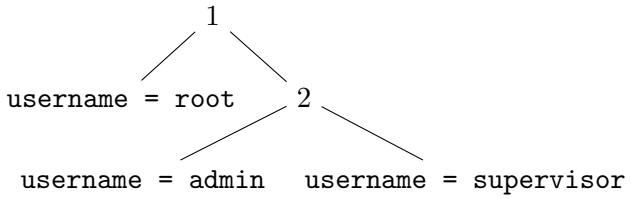


Figure 5.4: Choice tree

5.3 Variational Strings

In this section, I discuss two different types of variational strings and the rationale for choosing one of them. A straight-forward implementation of variational strings is to have a tree-like structure where each variant is a leaf, and the dimensions form the internal nodes. The alternatives of the choice become the children of the internal nodes. This representation of variational strings is called the choice-tree (also called a tag tree) [25, 20]. Listing 5.4 shows the type of a choice-tree. Selecting a variant using this representation amounts to traversing a path in the tree from the root to one leaf.

The choice-tree representation is not space efficient because the part of the string that does not contain any variation is duplicated in every leaf. For example, consider a properties file with a single key-value entry `username = root`. The value has been changed from `root` to `admin`, and again to `supervisor` in two different commits with ids 1 and 2 respectively, creating three versions of the file. Figure 5.4 shows this example using the choice-tree representation. As seen in the figure, the string `username =` is stored three times.

To eliminate this redundancy, strings common to different versions needs to be factored out. For this, we use the segment list [20] implementation of variational values. Segment list representation allows for the plain strings to be factored out and to have choices of only the changed strings. The type is given in listing 5.5.

Here, a variational string is a list of segments where each segment is either a plain string or a choice. This representation allows sharing of plain strings. Figure 5.5 shows

```
type VString = [Segment]

type Dim = Int

data Segment = Str String | Chc Dim VString VString
```

Listing 5.5: The type of a variational string as a segment list

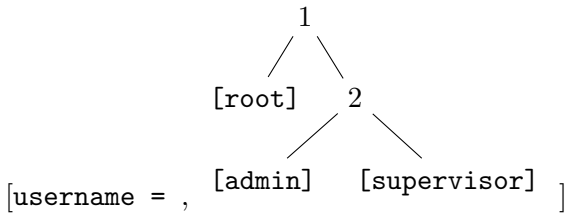


Figure 5.5: Segment list

the example in figure 5.4 represented using a segment list. Here, the plain string `username =` is stored only once and will be shared by all the versions. To select a variant, the choices are first resolved to string segments by traversing a path in the choice segment according to the view decision, and then all the string segments in the list are concatenated.

5.4 Matches

A match returned by vgrep is a list of commits and changes that the user has queried. Internally, this corresponds to a variational string. The result of a query can also be commit ids without the variational string match. Also, since GitQL allows variables in patterns as well as to refer to the match results, the type of a match should be able to accommodate both the types of variable binding. The values bound to these variables are then extracted from the result to either show as the output or to be reused in other match generators. Listing 5.6 shows the types for a match. A match generator returns a list of variational matches. A variational match consists of the matched variational string, its position, and all the variable bindings. An important point to be noted here is that a match statement in a GitQL query returns a list of matches. If this result is used as a source in another match statement, then the new pattern is matched against

```
type Matches = [VMatch]

data VMatch = VMatch MetaInfo VString QVarEnv

type MetaInfo = (Pos, DimEnv)

type Block  = Int

type Offset = Int

data Pos = P Block (Either Offset (Pos,Pos))
         | NoPos

type DimEnv = [(String,Dim)]

type QVarEnv = [(String, VMatch)]
```

Listing 5.6: The type of a match

each of the previous match result. This will produce a list of list of matches. Since users are only concerned with matches and not the specific source in which they were found, these nested lists are flattened to produce a single list of matches.

5.5 Vgrep

Each match statement in a GitQL query is a call to the function `vgrep`. The type of `vgrep` is:

```
vgrep :: Pattern -> VString -> Matches
```

The function `vgrep` takes a pattern, a variational string and finds all possible variational sub-strings that match the pattern. In the following, different scenarios that can occur during matching are explained. Consider a pattern P which does not contain a choice pattern.

Scenario 1: P matches a plain string.

In this scenario, vgrep matches the pattern similar to a regular expression search for strings. For example, if P is `public` and it is to be matched against a file that has no changes made then vgrep will simply look for all the occurrences of the word `public` in

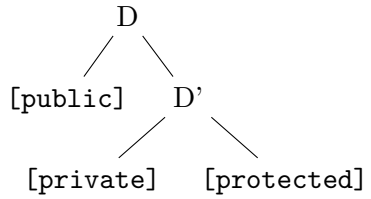


Figure 5.6: The tree structure of a choice segment

the file.

Scenario 2: P matches a choice.

The pattern is matched against both the alternatives of the choice there by producing a choice of matches. For example, if the pattern P is `public` and the Java file to be searched was modified in a commit where all the public variables were made private, the choices in the corresponding variational file would look similar to $D\langle \texttt{public}, \texttt{private} \rangle$. Since the pattern successfully matches only the left alternatives, the results will be of the form $D\langle \texttt{public}, \rangle$. The empty right alternative denotes that no match was found in the right alternative of the choice. In case an alternative contains a nested choice, the same pattern is matched in the alternatives of the nested choice. In the above example, say the variables were again changed from `private` to `protected`. The choices will then become $D\langle \texttt{public}, D'\langle \texttt{private}, \texttt{protected} \rangle \rangle$. When searching for the word `protected`, vgrep will match all the alternatives simultaneously and will yield the result $D\langle, D'\langle, \texttt{protected} \rangle \rangle$ preserving the entire change history of the string.

A choice can be viewed as binary tree of segments where the outermost choice is the root node, with all the alternatives at the leaves, and nested choices as inner nodes. The left alternative and the right alternative of the choice are in the left and right branches of the tree respectively. For example, the choice $D\langle \texttt{public}, D'\langle \texttt{private}, \texttt{protected} \rangle \rangle$ will have the tree structure shown in figure 5.6.

Matching a pattern against a choice therefore means matching all the leaves of the tree while preserving the path to it from the root. These matches do not depend on each other and therefore can be matched in parallel to speed up the process.

Scenario 3: P matches a combination of plain strings and choices.

This scenario has two different cases:

1 A plain string followed by a choice

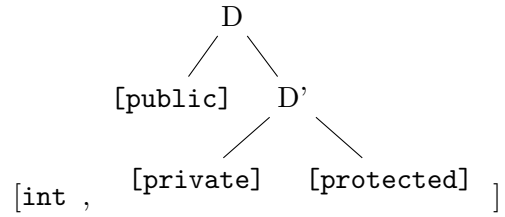


Figure 5.7: Segment list of scenario 3

2 A choice followed by a plain string or another choice

Assume that S is a plain string, and that $C\langle L, R\rangle$ is a choice where L and R may contain plain strings and/or choices.

Case 1: $S\ C\langle L, R\rangle$

In this case, P is first matched against S. If S is fully consumed, then the remainder pattern P$'$, is matched against the choice $C\langle L, R\rangle$ as in scenario 2. For example, assume P is `int private` in the example from scenario 2 where `public` was modified twice. A subsequence of the variational string that consists of these changes would be `int` $D\langle$`public`$, D'\langle$`private`$,$`protected`$\rangle\rangle$. The segment list of this expression is given in figure 5.7.

The plain string "`int `" is matched fully, and the remainder pattern `private` will be matched against the choice to return $D\langle, D'\langle$`private`$,\rangle\rangle$. The final result will be `int` $D\langle, D'\langle$`private`$,\rangle\rangle$.

Case 2: $C\langle L, R\rangle\ S$ or $C\langle L, R\rangle C'\langle L', R'\rangle$

The first step is similar to scenario 1 where the pattern is matched against the choice. The remainder pattern is matched against the plain string S or the choice C'. However, there could be two different remainder patterns as a result of matching against the alternatives of choice C. Consider the pattern `username =` which is used to match assignment statements of the variable `username`. If the variable had been renamed to `username` from `user`, the variational string in all the places where this assignment occurs will be "$D\langle$`user`$,$`username`$\rangle$ `= ...`".

The first step is to match P against the choice D. The left alternative is fully consumed and has the remainder pattern `name =`. The right alternative also is fully consumed, but the remainder pattern in this case is "` =`". There are two different remainder patterns to be matched against the segment that follows. If any one of the patterns is successfully

matched then the corresponding match from the choice is taken in the final result. In this example, the remainder pattern `=` matches successfully. Therefore, the final result will be $D\langle, \texttt{username}\rangle$ `=`. In the case where both the remainder patterns match successfully, the longest match is preferred.

To match such scenarios successfully, the pattern is split when segments of the variational string are consumed until the entire pattern is successfully matched. All the split patterns can be joined using the concatenation pattern i.e., P = $P_1P_2...P_n$ where P is split into n sub-patterns.

Vgrep matches choice patterns the way grep matches regular expressions. In the following, I will explain how choice patterns are matched against plain strings and choices.

Scenario 1: Plain Strings

Choice pattern are used to query the changes and therefore do not match plain strings. However, there is an exception to this rule. When a plain string is fully consumed by both the sub-patterns in a choice pattern, then it is included in the match result. For example, the choice pattern $d\langle\texttt{ab}, \texttt{ac}\rangle$ matches the plain string `a` because both the sub-patterns `ab` and `ac` successfully match the string `a` to produce the remainder pattern $d\langle\texttt{b}, \texttt{c}\rangle$ which is then matched against the next segment in the variational string.

Scenario 2: Choices

Choice patterns directly correspond to the choice calculus representation in the variational file. Therefore, matching a choice pattern means to find a choice in the variational file whose alternatives match the corresponding sub-patterns. The left and the right sub-patterns of a choice pattern match the left and the right alternatives of a choice, respectively. For a choice to be successfully matched, both the sub-patterns should be matched. If the sub-patterns do not match, then the choice pattern is matched against the choice alternatives to look for nested choices. For example, the choice pattern $d\langle\texttt{ab}, \texttt{cd}\rangle$ matches the choice $A\langle\texttt{ab}, \texttt{cd}\rangle$ successfully because both the sub-patterns match the respective choice alternatives. Consequently, the dimension variable d is bound to the dimension A. But for the choice $A\langle\texttt{xy}, B\langle\texttt{ab}, \texttt{cd}\rangle\rangle$, the right sub-pattern matches successfully and produces the result $B\langle, \texttt{cd}\rangle$ whereas the left sub-pattern fails. Therefore, choice A does not

match the choice pattern, and the successful match from the right alternative is discarded. The choice pattern is now matched against the plain string `xy` in the left alternative and against the choice $B\langle \texttt{ab}, \texttt{cd}\rangle$ in the right alternative. The match fails in the left alternative because it does not have a choice. The match in the right alternative succeeds because the choice $B\langle \texttt{ab}, \texttt{cd}\rangle$ exactly matches the pattern $d\langle \texttt{ab}, \texttt{cd}\rangle$. The final result from matching the choice $A\langle \texttt{xy}, B\langle \texttt{ab}, \texttt{cd}\rangle\rangle$ against the pattern $d\langle \texttt{ab}, \texttt{cd}\rangle$ is $A\langle, B\langle \texttt{ab}, \texttt{cd}\rangle\rangle$, and d is bound to the dimension B.

In the domain of querying a software repository, the users will not always know the exact change to look for. They would specify part of the string that was before the change or part of the string that was after the change. To enable such queries, the sub-patterns in a choice pattern can be variables or _ which not only matches any string, but also matches any choice in the context of a variational string. The examples for such choice patterns are shown in section 4.2.1 of chapter 4.

Chapter 6: Evaluation

To understand the kind of queries git users pose, we examined 2000 most viewed questions posted on Stack Overflow that were tagged with the label 'git'. These include questions about using git to perform basic version control operations such as commit, revert changes, create branches, merge branches, viewing changes introduced in commits, and so on. Of the 2000 questions, 96 questions (4.8%) were on obtaining history information such as commit logs, commit differences, commits in different branches, tags, and so on. Of these, 53 questions (55.2%) involved changes introduced in the commits. These questions in one way or the other were about the textual changes introduced in each of the commits. The remaining questions did not involve any textual changes introduced in commits and were about retrieving information such as tags in a repository, commit messages, names of the files changed in a commit, and so on. The questions that involved looking for textual changes are further divided into four categories based on the availability of git commands to solve the problem:

Category 1: Using git commands

Category 2: Using git commands along with manual work

Category 3: Using multiple git commands, external scripts, or tools

Category 4: Impossible to answer using existing git commands

The number of questions in each of the four categories is presented in table 6.1.

Category	**Questions**
Git commands	43 (81.13%)
Git commands and manual work	6 (11.32%)
Multiple git commands, external scripts, or tools	2 (3.77%)
Impossible to answer with git	2 (3.77%)

Table 6.1: Categories of Stack Overflow questions

Question	**Git Command**
View the change history of a file using Git versioning	`git log -p -- <filename>`
How to see the changes between two commits without commits in-between?	`git diff <commit1> <commit2>`
Compare local git branch with remote branch?	`git diff <local> <remote>`
How can I compare files from two different branches?	`git diff <branch1> <branch2>`
How can I view a git log of just one user's commits?	`git log --author="<name>"`
How to see the changes in a commit?	`git show <commit>`
Is there a quick git command to see an old version of a file?	`git show <commit>:<filename>`
How to retrieve a single file from specific revision in Git?	`git show <commit>:<filename>`
Finding diff between current and last versions?	`git diff`
See changes to a specific file using git	`git diff -- <filename>`
How to get the changes on a branch in git	`git log Head..<branch>`
How to check for changes on remote (origin) Git repository?	`git diff origin/master`
View a specific Git commit	`git show <commit>`
Shorthand for diff of git commit with its parent?	`git show <commit>`
How to view file history in Git?	`git log -p -- <filename>`
List all commits for a specific file	`git log -p --follow -- <filename>`
How can I calculate the number of lines changed between two commits in git?	`git diff --stat <commit1> <commit2>`
Git diff between given two tags	`git diff <tag1> <tag2>`
Using Git how do I find changes between local and remote	`git diff <local-branch> <remote-branch>`
Git diff file against its last change	`git log -p -1 <commit> -- <filename>`

How Do I run Git Log to see changes only for a specific branch?	`git log <branch-forked-off-of>`
How to show what a commit did?	`git show <commit>`
Find commit by hash sha in git	`git show <commit>`
Git: How to diff changed files versus previous versions after a pull?	`git diff HEAD@1 <filename>`
How do I see the commit differences between branches in git?	`git diff <branch1> <branch2>`
Git log to get commits only for a specific branch	`git log <branch>` or `git cherry -v <branch>`
How can I generate a git diff of what's changed since the last time I pulled?	`git diff <local> <remote>`
How can I get the diff between all the commits that occured between two dates with Git?	`git whatchanged`
Diff current working copy of a file with another branch's committed copy	`git diff <branch1>:<file1> <branch2>:<file2>`
Show diff between commits	`git diff <commit1> <commit2>`
Different commits between two branches	`git log <branch1>..<branch2>`
How to show first commit by 'git log'?	`git rev-list --max-parents=0 HEAD`
Get commit list between tags in git	`git log <tag1>..<tag2>`
Git diff between two different files	`git diff <branch1>:<file1> <branch2>:<file2>`
Git log of a single revision	`git show <commit>`
Git diff between current branch and master but not including unmerged master commits	`git diff master...<branch>`

How to find commits by a specific user in Git?	`git log --author="<author>"`
Difference between git HEAD and the current project state?	`git diff --cached`
How to check real git diff before merging from remote branch?	`git diff <local-branch> <remote-branch>`
How can I generate a diff for a single file between two branches in github	`git diff <branch1>:<file1> <branch2>:<file1>`
How to git log from all branches for the author at once?	`git log --author="<author>"`
How to get commit history for just one branch?	`git log --master`
Git diff between remote and local repo	`git diff <local-branch> <remote-branch>`

Table 6.2: Category 1 Questions

Questions from the first category are solved using `git diff`, `git show`, or `git log` and are displayed in table 6.2. In these questions, the users identify the commit before looking for the changes introduced by it either manually or using other commands and therefore, the actual intent of the users, as to what the high-level query is, is not clear. For a more effective evaluation, a thorough study on how users look through the history when given specific tasks related to querying history is required. Questions from the second category involve searching for strings within the changes introduced by the commits. These are solved specifically using `git log -S` or `git log -G`. Table 6.3 shows such questions with git commands and the corresponding GitQL patterns that can be used in a GitQL query that would return the exact matches. Therefore, users do not have to go through all the others changes introduced by the resulting commits. Table 6.4 shows the questions that require external scripts or tools. These are for viewing the entire history of a file in a more user-friendly way. Variational files consist of all the changes made to source files. In the case of chain commits, the order of the changes is preserved and therefore the users can look at the context in which these changes were made. However, this requires a user-friendly UI to effectively display the history and has been noted to be implemented as part of the future work. For the last category, the questions have no

Question	**Git Command**	**GitQL patterns**
How to grep (search) committed code in the git history?	`git log -S<string>`	`d<"<string>",x>`
Search all of Git history for a string?	`git log -S<string>`	`"<string>"`
How to grep Git commit diffs or contents for a certain word?	`git log -G<word>`	`d<x,"<word>"`
How do I "blame" a deleted line	`git log -S<line>`	`d<"<line>",x>`
How to search through all Git and Mercurial commits in the repository for a certain string?	`git log -S<string>`	`"<string>"`
Finding a Git commit that introduced a string in any branch	`git log -S <string> --source --all`	`d<x,"<string>">`

Table 6.3: Category 2 Questions

Question	Git Command
Git - go to particular revision	`git log -n1` and `git checkout <commit>`
Git - show history of a file?	GUI tool `gitk`

Table 6.4: Category 3 Questions

solutions. These are displayed in table 6.5 along with the reason why they cannot be solved currently.

6.1 Expressiveness

To evaluate the expressiveness of GitQL, we categorize some of the general patterns that can be queried on code changes and show how it can be achieved using GitQL and git commands.

1. Specific Changes: When users know a part of the change they are looking for and would like to know the commit that introduced those changes. For example, variable or function renames are some of the specific changes that users can query.

2. Chain Edits: When there have been multiple changes made to a particular text in a single branch. For example, changes made to a function body or multiple function renames.

Question	Reason
How to "git show" a merge commit with combined diff output even when every changed file agrees with one of the parents?	Three-way differencing is not present in git currently
List of all git commits?	Cannot query the changes of the commits that are unreachable from a branch

Table 6.5: Category 4 Questions

3. Conflicting Changes: When two branches consists of changes that affect the same part of a text in a file, then a conflict arises during the merge operation between the branches. These conflicts are resolved by the user by either selecting changes from the branches or making a different change altogether. Users might not look for conflicts intentionally, but when queried for the history of a particular text, any conflicting changes that were made in the past could be revealed. This is the case when features that have been developed in different branches are to be integrated and have one or more conflicts in it. When users query the changes made as part of the conflict resolution, they will find changes made in each of the branches. This view enables the users to review the conflict resolution.

Consider the history of a function `passwordMatched` that checks if the login password entered by a user is correct or not.

```
public Boolean passwordMatched(User user, String pwd)
{
    String hash = generateHash("MD5", pwd);
    String storedHash = getPassword(user);
    return hash.equals(storedHash);
}
```

Listing 6.1: Version 1 `passwordMatched`

```
public Boolean passwordMatched(User user, String pwd) throws
   NoSuchAlgorithmException
```

```
{
+    String hashingAlgo = "MD5";
-    String hash = generateHash(hashingAlgo, pwd);
+    String hash = generateHash("MD5", pwd);
     String storedHash = getPassword(user);
     return hash.equals(storedHash);
}
```

Listing 6.2: Version 2 passwordMatched

```
public Boolean passwordMatched(User user, String pwd) throws
    NoSuchAlgorithmException
{
-    String hashingAlgo = "MD5";
+    String hashingAlgo = "SHA-1";
     String hash = generateHash(hashingAlgo, pwd);
     String storedHash = getPassword(user);
     return hash.equals(storedHash);
}
```

Listing 6.3: Version 3 passwordMatched

```
public Boolean passwordMatched(User user, String pwd) throws
    NoSuchAlgorithmException
{
-    String hashingAlgo = "MD5";
+    String hashingAlgo = "SHA-256";
+    String pwdwithSalt = pwd+getSalt(user);
-    String hash = generateHash(hashingAlgo, pwd);
+    String hash = generateHash(hashingAlgo, pwdwithSalt);
     String storedHash = getPassword(user);
     return hash.equals(storedHash);
}
```

Listing 6.4: Version 4 passwordMatched

```
public Boolean passwordMatched(User user, String pwd) throws
    NoSuchAlgorithmException
{
     String hashingAlgo = "SHA-256";
     String pwdwithSalt = pwd+getSalt(user);
     String hash = generateHash(hashingAlgo, pwdwithSalt);
     String storedHash = getPassword(user);
     return hash.equals(storedHash);
}
```

Listing 6.5: Version 5 (Merged) `passwordMatched`

```
public Boolean passwordMatched(User user, String pwd) throws
    NoSuchAlgorithmException
{
-    String hashingAlgo = "SHA-256";
+    String hashingAlgo = "SHA-512";
     String pwdwithSalt = pwd+getSalt(user);
     String hash = generateHash(hashingAlgo, pwdwithSalt);
     String storedHash = getPassword(user);
     return hash.equals(storedHash);
}
```

Listing 6.6: Version 6 `passwordMatched`

The initial version of the function is in listing 6.1. The second version of the function in listing 6.2 has a new variable that is bound to the name of the hashing algorithm. In the subsequent versions the hash algorithms have been changed. Listing 6.3 and 6.4 are versions in two different branches that use two different hashing algorithms. These two branches were merged in version 6.4. The algorithm was again changed to `SHA-512` and is currently the latest implementation. These changes, although shown as consecutive changes in this example, could be separated by many number of commits. For each of the categories described above, a query in GitQL along with git command, if any, is listed as follows.

1. Specific Changes:

Query: In which commit was the latest “SHA-512” hashing introduced.

GitQL query: The following GitQL query will display the commit that matches any choice that has `SHA-512` in its right alternative. If the user is interested in the value before, then variable `x` can be listed alongside d.

```
d from m <- match d<x,"SHA-512"> in -f PasswordGen.java
```

Git command: Using the options `-S` or `-G`.

```
git log -S"SHA-512" -p -- PasswordGen.java
```

Or `git blame` which shows the latest commits that affected the specified line numbers.

```
git blame -L 3
git show commitId
```

2. Chain Edits:

Query: What was the hashing algorithm `MD5` changed to?

GitQL query: The result will contain changes from all the branches that were made to the text `MD5`.

```
x from m <- match d<"MD5",x> in -f PasswordGen.java
```

Git command: Git does not have a command that can be used to retrieve specific changes. The following command has to be used multiple times to retrieve all the changes.

```
git log -S"MD5" -p -- PasswordGen.java}
```

The above command will return the commit that added `MD5` and the commit that removed it. Users now have to again run the same command using the value it was changed to and so on until the latest change is known.

```
git log -S"SHA-1" -p -- PasswordGen.java
```

Git provides commands such as `rev-list` that lists commits in reverse chronological order, and `git-for-each-ref` that iterates over all the internal objects used in git to store commits, snapshots of files, tags, and so on. These commands, known as plumbing commands [19], are meant to be used to construct complex queries.

```
m′ from m <- match d<"MD5",x> in -f PasswordGen.Java,
         m′ <- match d′<y,z> in m
```

Listing 6.7: Reusing match results

They can be used in the UNIX way to compose different commands. The commands also allows certain UNIX tools to be combined. Plumbing commands are used in scripts and tools that require querying git internals.

3. Conflicting Changes:

Query: Retrieve all the algorithms that were used prior to `SHA-512`.

GitQL query: The following will result in all the hashing algorithm names that was present prior to `SHA-512` including `SHA-1` which was a conflicting change overridden in the merged version.

```
x from m <- match d<x,"SHA-512"> in -f PasswordGen.java
```

Git command: This is similar to the second category where the users have to run the command multiple times or use an external script but in the reverse chronological order.

From the above examples, it can be seen that using a single match statement, users can query version changes whereas, in git, multiple commands are required. Furthermore, in cases when git commands return more than one commit, the users have to browse through the changes introduced in each of the commits.

In GitQL, match results from one match statement can be used as the source in others, therefore, allowing users to build their queries incrementally. For example, the query in listing 6.7 first matches all the changes made to the string `MD5`. The result is then used as the source to match further changes made to it.

In git, it is not possible to express such queries. Users would have to write complex scripts or external programs that scan through the changes, identify the positions of those changes in the file, and look again in the history of the file if any commit affected those lines.

```
d, d′ from m <- match d<"foo()",_> in -f File1,
           m′ <- match d′<"bar()",_> in -f File2
```

Listing 6.8: Match multiple files

In GitQL, users can also write queries in which different patterns are used to match changes in different files. The query in listing 6.8 returns the commits in which function `foo` was renamed in *File1*, and the commits in which `bar` was renamed in *File2*. In git, users have to run the commands separately for each of the files.

GitQL can, therefore, express a wide range of queries, some of which are not possible using existing git commands and others require external scripts or users to browse through the changes manually.

Chapter 7: Conclusion

GitQL enables querying the change history through the underlying choice calculus representation as variational strings. It allows users to view the version history of files as a DAG of changes and query those changes without having to know the internal representation. Users can express queries that are either not possible using existing git commands, or involve multiple commands or external scripts. Choice patterns can be used to specify changes without having to go through the commit log. The result of a GitQL query is always a variational string that reflects the change history.

GitQL also provides a level of abstraction through the use of variables in patterns as well as building on previous match results. Users can begin with just one search statement and incrementally build a complex query by having multiple match statements for different patterns or nested queries.

The main focus of the thesis was to identify and describe various aspects of querying version histories and how they can be employed to aid the software development process. We introduced a general concept of variational pattern matching, implemented by vgrep. I hope this thesis will aid future research on variational pattern matching and version history.

As part of the future work, a user interface for the GitQL framework needs to be built. A user-friendly view of variational strings would increase understanding of what the match results mean. The choice encoding process currently encodes merges where conflict resolutions are simple in that the conflicting changes are discarded, and a new change is introduced. In the future, I plan on enhancing the tool to encode all the possible cases in a conflict resolution. Also, the choice encoding process currently does not perform efficiently for large repositories. Although it is a one-time process, it could be used frequently on multiple repositories in order to use GitQL, and therefore, the choice encoding tool needs to be improved to run on repositories of all sizes.

Bibliography

[1] Wyatt Allen. Variational parsing with the choice calculus. Master's thesis, Oregon State University, 2014.

[2] G. Canfora, L. Cerulo, and M. Di Penta. Tracking your changes: A language-independent approach. *IEEE Software*, 26(1):50–57, Jan 2009.

[3] CVS. `http://cvs.nongnu.org`.

[4] Darcs. `http://darcs.net`.

[5] Oege de Moor, David Lacey, and Eric Van Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1):15–35, 2003.

[6] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Transactions on Software Engineering and Methodology*, 25(1):7:1–7:34, December 2015.

[7] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, GPCE '13, pages 23–32, New York, NY, USA, 2013.

[8] B. Dhivya E. Kodhai. Detecting and investigating the source code changes using logical rules. In *2014 International Conference on Circuits, Power and Computing Technologies [ICCPCT-2014]*, pages 1603–1608, March 2014.

[9] Martin Erwig, Karl Smeltzer, and Keying Xu. A notation for non-linear program edits. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pages 205–206, July 2014.

[10] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Transactions on Softwware Engineering and Methodology*, 21(1):6:1–6:27, December 2011.

[11] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov 2007.

[12] Git. `https://git-scm.com`.

[13] GitHub. `http://github.com`.

[14] Verónica Uquillas Gómez, Andy Kellens, Johan Brichau, and Theo D'Hondt. Time warp, an approach for reasoning over system histories. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 79–88, New York, NY, USA, 2009.

[15] Hadoop. `http://hadoop.apache.org`.

[16] Abram Hindle and Daniel M. German. Scql: A formal model and a query language for source control repositories. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005.

[17] Alex Loh and Miryung Kim. Lsdiff: A program differencing tool to identify systematic structural differences. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 263–266, New York, NY, USA, 2010.

[18] Mercurial. `https://www.mercurial-scm.org`.

[19] Git plumbing command. `https://git-scm.com/book/no-nb/v1/Git-Internals-Plumbing-and-Porcelain`.

[20] Karl Smeltzer and Martin Erwig. Variational lists: Comparisons and design guidelines. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development*, FOSD 2017, pages 31–40, New York, NY, USA, 2017.

[21] 2015 Stackoverflow developer survey. `https://insights.stackoverflow.com/survey/2015`.

[22] R. Stevens and C. D. Roover. Querying the history of software projects using qwalkeko. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 585–588, Sept 2014.

[23] Reinout Stevens. A declarative foundation for comprehensive history querying. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 907–910, Piscataway, NJ, USA, 2015.

[24] Subversion. `https://subversion.apache.org`.

[25] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 213–226, New York, NY, USA, 2014. ACM.

[26] Eric Walkingshaw and Klaus Ostermann. Projectional editing of variational software. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 29–38, New York, NY, USA, 2014. ACM.

[27] Thomas Zimmermann. Fine-grained processing of cvs archives with apfel. In *Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '06, pages 16–20, New York, NY, USA, 2006.