Git Query Language and Variational Grep

*

ABSTRACT

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Design, Languages, Theory

Keywords

variation, version history, pattern matching, grep

GQL is language to query the history of git repositories. Before going to the GQL, let us first see the need to query the history and how it is done till now is.

One of the advantages of using version control systems (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. These changes help users or programmers understand the workings of the current version because they can see what changed from the previous version. In the case of software bugs, users can quickly narrow down the part of code that caused it by identifying what changed between the version that worked correctly and the current version. Hence, the history of software plays a vital role in its maintenance.

The history is important even during the development phase. A VCS makes it easier for the developers to try different implementations and choose one among them. This is either done by having a different implementation in a separate branch or by reverting to a previous version in the same branch in the case the latest implementation did not work. Querying even a shorter history helps the programmer to look at the changes she made and compare them.

*Does NOT produce the permission block, copyright information nor page numbering. For use with `sig-alternate-05-2015.cls`. Supported by ACM.

Searching for a word or a sentence in a single file is an easy task since there are many tools and commands available. One can look for exact words or a pattern of words in which case a regular expression is used to specify the pattern. A regular expression is a sequence of characters which acts as a syntax for the matching texts.

The language for regular expressions consists of literal characters including the empty string. Strings are obtained by concatenating multiple characters. The regular expression also consists of alternation and Kleene star as described below

1 Alternation: R|S denotes either R or S , both of which are regular expressions, to be matched. For example, the regular expression `" "(a|an|the)" "` matches any of the articles. The space before and after is to make sure they are separate words and not substrings of other words.

2 Kleene Star: R* denotes zero or more occurrences of the pattern R. For example, `a(b)*` matches text that starts with `a` followed by zero or more number of `b`'s.

For example, the regular expression $[A-Z][a-z]*$ matches the text that starts with an upper case letter followed by zero or more lowercase letters. Grep is a Unix tool that can search a file using regular expression.

In a file that is version controlled, searching involves looking into a specific 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 return the ones that contain the pattern. This brute force approach is straightforward and easy to implement. However, it is not efficient when there are a huge 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 allow users to search using regular expressions.

In cases where the users want to query the changes and not the entire file, git commands do not suffice. Consider a case where the user is looking for a specific change, that is,

some string s changed to t. Git consists of commands that can be used to query what was added or removed in each version. Such a command would return all the versions that added s and all the versions that removed t instead of just the version that replaced s with t. Git also has commands that can track the changes made to a range of lines in the file. However, these are not accurate sometimes since the line numbers keep changing due to addition and deletion of lines. Most of these commands are complex and sometimes require external scripts as well.

Consider a scenario where all the changes made to a function needs to be looked into for debugging purposes. A search for the function name will return all the versions that contain the function. What we need in this case are all the versions that have some changes made to the function. Git log with the option -L searches the history of a range of lines specified by a start and an end. But this range will change when lines are added or deleted around the function, and therefore not all the changes are captured.

Hence there is a need for a language that is simple, and that enables users not just to look for patterns but also to look for changes in the history of software repositories.

In GQL, users can query the history similar to grep by using regular expressions. An important feature is, however, to query specific changes. In GQL this can be achieved using the "Choice Patterns." In a choice pattern, users can specify a pattern for the text in the previous version and another pattern for the text in the later version. Therefore, retrieving only the versions that made such changes. For example the pattern to find the version where a function *foo* was renamed to *bar* is $D\langle foo, bar\rangle$. The syntax for choice pattern is $D\langle Pattern - before, Pattern - after\rangle$. The character D is called the dimension which is explained in further sections.

[Cite paper, Eric's Ph.D. thesis, leadGTTSE tutorial] To achieve this feature, we represent the history of a repository using a formal language for variation called the choice calculus [Cite]. The choice calculus is a generic language that can be applied to an arbitrary object language. The choice calculus consists of *"choices"* to denote variations. The syntax of a choice is $D\langle A, B, ..\rangle$ where A,B and so on are called the alternatives. The alternatives could be terms in the object language or choices themselves. Each of the alternatives in a choice leads to a different version. Therefore, a choice represents all possible variations of an expression in the object language from which one can be selected. Using the choice calculus representation, we get a simple tree model that allows to focus on the changes that lead to newer versions.

Consider the following example of two different implementations of the function `twice` in Haskell.

```
twice x = x+x

twice x = 2*x
```

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

`twice x =` $D\langle$`x+x, 2*x`$\rangle$

Here, both the alternatives of the choice are terms in the object language, in this case, Haskell. A choice is bound to a dimension D. Two choices can have same or different dimension names. To understand this better, in the above example, consider two more implementations where the variable is named as `y` and `z`. Therefore, we now have four additional implementations of the function `twice`.

```
twice y = y+y

twice y = 2*y

twice z = z+z

twice z = z*z
```

The choice expression now has two dimensions, each representing a variation in the variable name and function body respectively, therefore, capturing the fact that these are variations in two different aspects.

`twice` $B\langle$`x, y, z`$\rangle$ =
$A\langle B\langle$`x, y, z`$\rangle$`+`$B\langle$`x, y, z`$\rangle$`, 2*`$B\langle$`x, y, z`$\rangle\rangle$

Two choices with dimension name B are nested in the choice with dimension name A thereby representing variations in 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.

From a variational program consisting of choices, a concrete program variant can be obtained 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*. A selection is a mapping from a dimension name to an alternative. The set of selections required to obtain a concrete program is called a *decision*.

From the choice expression in the above example, to obtain the variant of `twice` which has '`y`' as the variable and implemented using '`+`', we have to make two selections; one for the choice of variable names and the other for the choice of the function body. A selection is denoted by D.1, D.2 and so on where D is the dimension name and 1,2,..n are the indexes of the n alternatives that occur in the same order. The decision {A.1, B.2} will select the first alternative from the choice with dimension name A and the second alternative from the choices with dimension name B and produce the following concrete program in the object language.

```
twice y = y+y
```

From the above process of selection, it can be observed that B.2 selects the second alternative from all the three choices that are named B. The dimension names, therefore, synchronize the changes that are made in different places but are all part of the same variation.

To represent the version history of a file using the choice calculus representation, we use the choice edit model[Cite] which is an application of 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. In the choice edit

model, making an edit to a program is viewed as introducing variation into the program. Whenever a program part P is changed to Q, a choice between P and Q is introduced. Here, a choice will have only two alternatives - the left and the right alternative; the left alternative consists of the old value and the right alternative consists of the new value. Consequently, selection can be expressed as D.L and D.R for left and right alternatives respectively. The object language, in this case, is simply a string of characters because we are capturing the textual edits and do not care if the program behavior changes or not.

Consider the following C program P which consists of a function *f*. This is the first version of the program. No changes have been made to it and therefore the variational program VP is same as P.

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

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

A programmer makes the following sequence of edits on *f*. After each edit, choices are added in the variational program VP.

Edit 1: Change the function argument to `c`. A choice with dimension name, say A, between `a` and `c` is introduced in all the places where the edit is made.

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

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

Edit 2: Assign the value 1 to the variable `b`. A choice with a different dimension name, say B, between an empty string and " *= 1*" is introduced. This edit is an insert operation and therefore there is no old value. Similarly, for a delete operation there is no new value and therefore, the choice that represents a delete operation will have an empty string in the right alternative.

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

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

Edit 3: Change the function argument again to `d`. A new choice with dimension name, say C, is introduced. This edit is changing the string that has already been changed previously and therefore is nested in the right alternative of A. Such edits are called chain edits.

```
P3 = int f(int d)
    {int b = 1; return d+b;}

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

For each edit operation, a new choice is introduced. To synchronize the edits made in different places, the choices are given the same dimension name. The variational program VP3 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. The latest version can be obtained by selecting the right alternative from all the choices in VP3. The programmer can select any variant by making appropriate selections. For example, the decision {A.R,B.L,C.R} will generate a variant that consists of Edit 1 and Edit 3 but not Edit 2 which is nothing but a selective undo. Following is the program variant obtained after applying the decision {A.R,B.L,C.R}

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

Version controlling a file is same as program editing but the edits are usually larger. 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 to a file, choices can be added like the way it is done in the choice edit model. All the choices corresponding to a commit will have the same dimension. This way we can identify all the changes made by a specific commit. The selective undo and redo operations in the choice edit model correspond to git's revert and cherry-pick commands. Using the choice edit model, we create a variational file that contains the entire change history of the file.

A part of a variational file is shown in the listing 1. It shows a function initially named as "contains" is renamed to "elem" and then again to "present." θ is used to differentiate between the choice construct and the object language. The dimension names used here are integers and start from one. These dimension names also capture the temporal aspect of the commits. For example, choices with dimension name "1" are introduced before the choices with dimension name "2". It also implies that the changes encoded by the choices with dimension name "1" were made before the changes encoded by the choices with dimension name "2". Queries in GQL are run on the variational file and therefore we first need to generate a variational file for each file that is being version controlled. This is done as a one-time step by a separate tool. The tool starts with the first version of the file and copies the contents into the variational file as is. From the second version onwards, it computes the differences and encodes them into choices. The difference between the previous and the current version of the file is found using the longest common subsequence algorithm [Cite]. The tool iterates through all the versions in a chronological order and produces a variational file in the end.

Listing 1: Encoding of version history of a Haskell program file

```
...
θ1⟨containsθ,θ2⟨elemθ,presentθ⟩θ⟩ :: Eq a => a ->
     Tree a -> Bool
θ1⟨containsθ,θ2⟨elemθ,presentθ⟩θ⟩ x Leaf = False
θ1⟨containsθ,θ2⟨elemθ,presentθ⟩θ⟩ x (Node y l r)
  = x == y ||
    θ1⟨containsθ,θ2⟨elemθ,presentθ⟩θ⟩ x l ||
    θ1⟨containsθ,θ2⟨elemθ,presentθ⟩θ⟩ x r
```

GQL provides various constructs that can be used by the programmer to query the history. The most important and

the one which forms the crux of the language is the `vgrep` construct. The variational grep or `vgrep` takes a pattern and a variational file, looks for the pattern in it and returns all the occurrences of the pattern. The result of the query is also variational and stores the necessary context information. Following is the syntax of `vgrep`.

```
vgrep <pattern> <variational_file>
```

The patterns in GQL extends the basic regular expression language to include the choice patterns. Following is the syntax of a pattern.

```
<pattern> ::= <character>
            | <anyCharacter>
            | <pattern> <pattern>
            | <pattern> "|" <pattern>
            | <dimension> "⟨" <pattern> "," <
                pattern> "⟩"
            | "$" <string>
            | "(" <pattern> ")"
character    ::= <ascii_character_set>
anyCharacter ::= "."
dimension    ::= Integer | string
string       ::= character string | ϵ
```

A character in a pattern can include any ASCII character. However, since certain symbols are used to denote other patterns, these symbols, called metacharacters, need to be escaped using a backward slash in order to search them. Otherwise, they will be treated as metacharacters. For example, the metacharacter dot (.) is used to specify any character. So, if a user is looking for a pattern that consists of a full stop, then \. should be specified in the pattern. This way the search algorithm can correctly interpret the intended use of such symbols.

A sequence of patterns can be formed by concatenating one pattern with the other. This way, the user could search for a string `ab` or a string followed by a choice pattern `int` $1\langle \texttt{foo}, \texttt{bar} \rangle$ and so on.

The choice pattern, as explained previously can be used to specify the changes. The dimension in a choice pattern can be an integer or a choice variable. In the case of an integer, `vgrep` looks for a choice with the exact dimension name. This corresponds to searching for the change made by a particular commit. For example, the pattern $1\langle \texttt{foo}, \texttt{bar} \rangle$ matches $1\langle \texttt{foo}, \texttt{bar} \rangle$ in the variational file but not $2\langle \texttt{foo}, \texttt{bar} \rangle$ since the dimension name differs. If a dimension variable is used, then `vgrep` only searches for choices based on the patterns. If both the patterns match, then the dimension variable is bound to the dimension name of the matched choice. For example, the pattern $d\langle \texttt{foo}, \texttt{bar} \rangle$ matches both $1\langle \texttt{foo}, \texttt{bar} \rangle$ and $2\langle \texttt{foo}, \texttt{bar} \rangle$ where d is bound to 1 and 2 respectively. Dimension variables are useful when the dimension name is unknown or when a pattern could occur in multiple versions. Each match returned by `vgrep` will have the value to which the dimension variable is bound.

TODO: Should we specify commit id instead of dimension name?

It can be argued that, the alternatives of a choice can be matched using the alternation. pattern For example, the pattern `foo|bar` matches the choice $1\langle \texttt{foo}, \texttt{bar} \rangle$(although they are two different matches; one for "foo" and the other for "bar"). The semantics of the alternation pattern do not consider the position of the matches which means that either of the two patterns could occur first. Therefore the pattern `foo|bar` also matches $1\langle \texttt{bar}, \texttt{foo} \rangle$. But the semantics of querying the change $1\langle \texttt{foo}, \texttt{bar} \rangle$ is different from that of $1\langle \texttt{bar}, \texttt{foo} \rangle$. In the former choice, it means "foo" is replaced by "bar" and in the latter, it means "bar" is replaced by "foo". The choice pattern captures this semantics and produces intended results.

Query variables in a pattern can be used to refer to a part of the query result. This is specifically for choice patterns when the only user knows the pattern for either the old value or the new value in the variational file but wants the change history of that value. By specifying query variables in the right or left alternative of the choice pattern, the user gets the new value or the old value respectively. For example, in the pattern $d\langle \texttt{foo}, \texttt{\$q1} \rangle$, the query variable $q1 is bound to all changes made to the text "foo" in the variational file since the chain edits are encoded in the right alternative of the choice. These values, to which the query variables are bound, can be simply viewed or further queried.

Apart from `vgrep`, GQL provides the following constructs.

`count` :: To count the number of matches returned by a query.

`countDim` :: To count the number of dimensions that matched. This is used in the case of choice patterns that uses dimension variables. It essentially counts the number of versions in which a match was found.

`show` :: Pretty print the result of a query.

`oR` :: Takes two `vgrep` results and return either of the successful matches

`anD` :: Takes two `vgrep` results and returns them only if both have successful matches.

`filter` :: Takes a predicate based on which it filters out the matches.

`subQuery` :: Takes a pattern and a list of matches which were obtained from other queries and looks for the pattern in those matches. This construct allows the user to essentially nest GQL queries

[TODO : add concrete syntax for the above list of constructs]