AN ABSTRACT OF THE THESIS OF

Aarti Chabra for the degree of Master of Science in Computer Science presented on December 13, 2011.

Title: Structured Representation of Composite Software Changes

Abstract approved: _____

Martin Erwig

In a software development cycle, programs go through many iterations. Identifying and understanding program changes is a tedious but necessary task for programmers, especially when software is developed in a collaborative environment. Existing tools used by the programmers either lack in finding the structural differences, or report the differences as atomic changes, such as updates of individual syntax tree nodes.

Programmers frequently use program restructuring techniques, such as refactorings that are composed of several individual atomic changes. Current version differencing tools omit these high-level changes, reporting just the set of individual atomic changes. When a large number of refactorings are performed, the number of reported atomic changes is very large. As a result, it will be very difficult to understand the program differences.

This problem can be addressed by reporting the program differences as composite changes, thereby saving programmers the effort of navigating through the individual atomic changes. This thesis proposes a methodology to explore the atomic changes reported by existing version differencing tools to infer composite changes. First, we will illustrate the different approaches that can be used for representing object language program differences using a variation representation. Next we will

present the process of composite change inference from the structured representation of atomic changes. This process describes patterns that specify the expected structure of an expression corresponding to each composite change that has to be inferred. The information in patterns is then used to design the change inference algorithm. The composite changes inferred from a given expression are annotated in the expression, allowing the changes to be reported as desired.

Structured Representation of Composite Software Changes

by

Aarti Chabra

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 13, 2011
Commencement June 2012

Master of Science thesis of Aarti Chabra presented on December 13, 2011.

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.

Aarti Chabra, Author

ACKNOWLEDGEMENTS

I express my sincere gratitude to Dr. Erwig for his support and guidance that made this thesis possible. I am really thankful for his invaluable time and feedback that helped me in writing this thesis.

I sincerely thank Dr. Burnett and Dr. Budd for helping me with my defense.

I thank my family for their love and support, especially my husband who motivated me to pursue further studies.

I am very thankful to my research group members, Eric, Duc, Rahul, Sheng, Tim and Chris, for their help. Lastly, I thank all my dear friends, especially Maddy, SS, NN, AT, Harsha for making my graduate life memorable.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

DEDICATION

To Aru, for his encouragement and support.

Chapter 1 – Introduction

1.1 Introduction

Programs undergo changes throughout the software development process. These changes can be attributed to various operations, such as adding new functionality, fixing bugs, program restructuring etc. These changes can be categorized as atomic or composite. An *atomic change* is a single independent change, such as adding, deleting or updating an expression. A *composite change* is composed of several atomic changes that are related to each other, such as refactorings. Generally, a software is developed in a collaborative environment, where more than one programmer may work on a single program. For an efficient contribution of a programmer, it is necessary for him/her to understand how the program has evolved.

The above-mentioned necessity has motivated the research in the field of software version management and has resulted in the culmination of various version differencing tools. These tools help the programmers by finding and reporting the differences between the two versions of a program. One category of version differencing tools, such as *diff* [30] finds line based differences between the two files. These tools are helpful when comparing the two versions of files that contain textual content. However, for program files these tools are unable to report differences in terms of structural differences.

The limitation of line based differencing tool for finding program differences was addressed by the abstract syntax tree based tools [18][5][31]. These tools compare the abstract syntax trees of the two program versions and report the differences in terms of the nodes that have been added, deleted or updated. Although these tools report structural differences between the program versions, these

tools suffer from drawbacks. Firstly, these tools lack structure for representing changes. Secondly, the program differences are reported as atomic changes. As a result, a programmer has to go through the entire set of the atomic changes to understand the differences between the two program versions. The process of understanding the changes becomes time consuming and tedious when the set of reported atomic changes is large.

The limitations discussed above have motivated this research to design a system that provides a structured representation to the program differences and reports these differences from a high-level perspective. We do not intend to create a new version differencing tool, as a vast research has already been done in this area. Instead, we want to analyze the differences reported by the existing version differencing tools to detect composite changes in order to explain the changes from a high-level perspective. We hope that with such a system, we will be able to reduce the time and effort that a programmer spends on understanding the program differences.

We first illustrate the representation of the atomic changes in an object language program using a variation representation. A variation representation imparts structure to the differences between the two object language programs. The structured representation of the atomic changes would allow these changes to be analyzed for inferring composite changes. We design a new language called *variational object language* that combines the object language and the variation representation constructs. The variational object language expressions representing the atomic changes are called *variation expressions*.

Next, we assume that a tool exists that uses any of the existing AST based version differencing tools [5] to find the differences between the two program versions and reports the atomic changes as variation expressions.

With the program differences available as variation expressions, we design the composite change inference process that explore these variation expressions to infer composite changes. First, for each change that we intend to infer, we study the structure of the variation expression resulting from that

change, along with the atomic changes that it contains. This information is used to specify variation expression templates for each composite change. These templates guide the design of composite change inference algorithm that analyzes a given variation expression and compares it with variation expression templates that have similar structure. A match between a given variation expression and the expected template implies that the expression corresponds to the change the template is associated with. As a result, the composite change information is annotated in the given expression. The annotated information allows the changes to be reported as desired.

To design the system mentioned above, we select lambda calculus as the object language and choice calculus as the variation representation. The advantages offered by choice calculus such as modularity and structuredness, make it an ideal candidate for the variation representation.

We expect that the above-mentioned approach will result in an efficient management of the program changes. Representing the changes using variation representation will impart structure on the changes allowing these to be analyzed and transformed if required. Reporting the changes as composite changes instead of atomic changes will make it easier and time efficient for the programmer to understand the program differences.

The next section describes the problem associated with understanding the program changes. Section 1.3 discusses the existing systems of representing and reporting changes along with their limitations. Section 1.4 uses an example to illustrate how the existing systems can be improved. In Section 1.5, we describe the approach that is used to design the intended system for representing and reporting the program changes. The results that are expected with the new approach are described in Section 1.6. The last section provides a brief outline of the chapters that follow.

1.2 Problem Statement

Consider a programmer who works on a program p_1 and updates the program resulting in another program version p_2. Suppose, other programmers also contribute to the same program and their changes result in different versions of the program. At any point, a programmer can use a version differencing tool to find the differences between any two program versions, p_i and p_j. The program differences are inferred in terms of atomic changes and are reported in either a textual format or highlighted using color-coding.

The existing tools report only the atomic changes between the two versions and these changes are easy to understand. However, if a large number of atomic changes are reported, then it becomes tedious for a programmer to navigate through all the atomic changes. As a result, understanding the differences in programs becomes a time consuming task.

Suppose, the information about the program differences is required by some other software tool for tasks such as testing the equality of two sets of program differences. The changes represented in an unstructured way cannot be used by other tools. Hence, for any tasks that requires analysis of program changes, these changes should be represented in a structured format.

1.3 Existing systems

The existing systems for variation management include line based [26] or abstract syntax tree based [18] [40] [5] [31] version differencing tools. The first limitation of these tools is that they do not provide a structured representation to the program differences. Most of these tools either represent the changes as text or highlight the changes. This makes it difficult to further analyze these changes. The second limitation of these tools is that the program differences are represented as atomic changes that can result in a large set of reported changes.

The refactoring detection tools [7][24], are either detect very limited refactorings or require human intervention. The tools [15] [8] [38] explore the version differences to infer refactorings, but are limited to the refactorings related to class structure or methods, or cannot infer multiple refactorings. Addtionally, these tools do not detect the exceptions to the inferred refactorings.

Kim et al. [22] overcome most of the limitations described above, but their approach is different from our approach in several ways. They represent the program versions in terms of facts, which does not provide complete contextual information. These facts are then used to compute the fact-level differences. These differences along with the original fact-bases are used to infer the logic rules that determine the high-level changes. The logic rule inference can require multiple references to the original fact-bases. The inference algorithm matches the inferred rules with the fact-level differences for detecting refactorings.

Instead of building a separate database of facts, we represent the changes using choice calculus that represents differences at the precise locations and no information is lost. Instead of logic rules, our approach defines variation patterns that includes the information about the expected constructs and the changes, for each composite change that has to be inferred. The inference algorithm in our approach also matches the variation patterns with the version differences, but as these differences are represented within the expression this does not require any additional references. Additionally, the inferred composite changes and the exception cases are annotated in the expression itself, allowing the flexibility to report these changes as needed.

1.4 Proposed Solution

Consider the lambda calculus expression;

$$e_{old} \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 2}$$

An update of 2 to 3 when performed on the expression e_{old} results in expression e_{new} as follows:

$$e_{new} \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 3}$$

The version differencing tools will report the differences between e_{old} and e_{new} as an atomic change of 2 to 3 in the scope of the definition of function f. Suppose, function renaming is performed on e_{old}, which results in updating of f to g in the function name as well as for each occurrence of f in the scope of the function definition of f. As a result, the expression e_{old} is changed to the expression e'_{new} as follows:

$$e'_{new} \equiv \texttt{let g=}\lambda\texttt{x.succ x in g 2}$$

Although the differences between e_{old} and e'_{new} are result of function renaming, the existing version differencing tools will report the differences as two atomic changes. First, as an update of f to g in the function name and the second update in the function scope.

Suppose, these differences are represented in a structured format as follows:

$$e_D \equiv \texttt{let } D\langle\texttt{f,g}\rangle\texttt{=}\lambda\texttt{x.succ x in } D\langle\texttt{f,g}\rangle\texttt{ 2}$$

The above expression uses the construct $D\langle\texttt{f,g}\rangle$, let us assume that this construct is a change expression that contains an ordered pair of alternatives. The value f is the old value that is updated to the value g. The above expression contains two change expressions defined within the expression at a fine-grained level.

Suppose, instead of the reporting the program differences as atomic changes, a version differencing tool could report the differences in terms of the composite changes that have been done to the program. For example, instead of reporting the changes between e_{old} and e'_{new} as the expression

e_D, the following expression is reported as:

$$r \equiv \mathit{RenameF}(\texttt{f},\texttt{g})\ \textbf{in}\ \texttt{let}\ D\langle \texttt{f},\texttt{g}\rangle{=}\lambda \texttt{x.succ x in}\ D\langle \texttt{f},\texttt{g}\rangle\ \texttt{2}$$

The above expression clearly states that a function renaming has been performed where the variable is changed from `f` to `g` in the expression $\texttt{let}\ D\langle \texttt{f},\texttt{g}\rangle{=}\lambda \texttt{x.succ x in}\ D\langle \texttt{f},\texttt{g}\rangle\ \texttt{2}$.

Representing the changes using a variation representation allows the analysis of these changes. And reporting the differences as composite changes as shown above, provides a more informative explanation about the changes that has been performed. The above-mention system of representing and reporting the program differences can potentially reduce the programmer's effort for understanding these differences.

The above discussion suggests that a new system for representing and reporting the software variations is required that adds the following improvements to the existing system:

1. Provides a structured format to the changes.

2. Reports the program differences as the composite changes that have been performed.

1.5 Thesis Approach

This thesis aims to design a system that incorporates the improvements suggested in the previous section. The first improvement of structured representation of the program differences is achieved by using a variation representation. This thesis illustrates the representation of differences in an lambda calculus using the choice calculus by designing a new language called *variational lambda calculus*. Although variational lambda calculus has been introduced earlier for static analysis of variational programs [6], the syntax can represent only limited variations. Chapter 4 presents the different approaches for designing a variational object language that uses choice calculus as varia-

tion representation. These approaches are explored by designing the syntax of variational lambda calculus that can provide most appropriate representation for atomic and composite changes.

As this thesis does not intend to design a version differencing tool, we make an assumption that there exists a tool that represents the differences reported by a version differencing tool using variational lambda calculus. Hence, the further explanation of our approach assumes that the program differences as variation expressions described in variational lambda calculus are available.

The second improvement suggested in the previous section is incorporated by designing a composite change inference process. This process is implemented in several steps. First, the refactorings that have to be inferred are selected and the expected structure of variation expressions corresponding to the selected changes is specified as *variation pattern*.

Next, the information in variation patterns is used to design *change inference algorithm* in Haskell. Depending on the structure of the given variation expression, the algorithm selects the composite changes that are possible. Then the algorithm checks if a given expression matches the expected variation expression and the atomic changes specified for the selected composite changes. A match indicates that the expression corresponds to a composite change, and the change information is annotated in the expression. A variation expression with the refactoring annotations is called *refactoring annotated expression*. The annotated change information provides complete information about the changes, including the name of the composite change, the old and the new value. If a match is not found, then the variation expression is left unchanged.

In addition to the improvements suggested in the previous section, we add another improvement of reporting exception cases. An exception is an anomaly to the composite change. Suppose a variation expression suggests the renaming of a function from `f` to `g`, this means that the name of the function is changed and all the occurrences of `f` in the scope are also changed to `g`. An exception to function renaming will be an occurrence of `f` in the scope of the function in the new version.

As discussed above, this thesis aims to design a system that offers the following features:

1. Represents the atomic and the composite differences between two programs in a structured manner.

2. Reports the program differences as composite changes that provides a high-level view of the changes.

3. Provides the necessary explanation for the inferred changes with refactoring annotations.

4. Provides additional information such as exceptions to the reported changes.

1.6 Expected Results

The following are the results that are expected from the system designed in this thesis:

1. The changes (atomic and composite) in lambda calculus will be represented in a structured manner.

2. The change inference algorithms will be able to detect the composite changes for lambda calculus from the variation expressions.

3. The detected composite changes will be annotated in the variation expression that provides the flexibility to report the changes and to perform any further analysis on the inferred composite changes.

4. The annotated variation expression will provide all the necessary explanation about the changes.

1.7 Outline

This thesis is structured as follows:

- Chapter 2 discusses the existing work done in the related fields. This chapter includes the work done in the area of variation management, and identification of refactorings.
- Chapter 3 describes the choice calculus [9] that is a vital part of this thesis. The concepts and the definitions that are extensively used in the later part of this thesis are briefly discussed here.
- Chapter 4 introduces various approaches that can be used for designing a variational lambda calculus and lists out the advantages and limitations of each of these. One of these representations is selected as the final syntax to be used in the process of inferring changes.
- Chapter 5 explains the process of composite change inference using variation patterns and change inference algorithms.
- Chapter 6 describes the conclusions made by our research and the scope of future work that can be done using this research.

Chapter 2 – Literature Review

2.1 Introduction

This chapter discusses the related research work that has been reviewed for this thesis. The research done in the field of variation management is discussed in Section 2.2. Section 2.3 discusses the concept of refactorings, its identification and benefits. In Section 2.4 we discuss different refactoring tools that identify potential refactorings. Section 2.5 discusses the refactoring tools that explore the version differences to find the related changes. These tools aim to provide a better explanation of the software changes. In Section 2.6 we describe the differences of our approach from the research described in earlier sections.

2.2 Variation Management

This section discusses different researches that have been developed to deal with software variations. We start by discussing the tools that handle variations in a single dimension. These tools include the version differencing tools that manage variations with respect to time. Next, we discuss the tools that can manage multi-dimensional variations, such as CPP etc.

One of the earlier research works done in finding files differences is by Hunt et al. [26]. They present a *diff* program that reports the file differences by finding the minimum number of changes that are required for matching one file to another. The underlying algorithm is based on the solution for longest common subsequence problem. The algorithm uses dynamic programming and improves the efficiency to $O(mn)$ by using the concept of essential candidates [17], where m and n

are the lengths of the two sequences. The file differences are reported as delete, change, and append operations. Later, Myers [30] uses a greedy approach to improve the efficiency of the *diff* algorithm to O(ND), where N is the sum of the lengths of the two subsequences and D is the size of the edit script for the two sequences.

Horwitz [18] points out the limitation of line-based file comparison tools that do not take program structure into account. She addresses this limitation by creating program representation graphs for the two versions *New* and *Old* of the program, and using a partitioning algorithm [41] that divides the components into different partitions on the basis of their behaviors. The changed components are categorized as either textual change or semantic change. A component c of *New* is categorized as a semantic change if either of the following two conditions are satisfied:

- There is no component in *old* corresponding to c.
- Different sequence of values are produced at c in *old* than at *new*.

Horwitz illustrates the difference between the textual and semantic change with the following example:

old	*New*	*New*
`x:=0`	`x:=0`	`a:=0` $\leftarrow$ *TEXTUAL*
`if P then`	`if P then`	`if P then`
`x:=1`	`x:=2` $\leftarrow$ *SEMANTIC*	`a:=1` $\leftarrow$ *TEXTUAL*
`fi`	`fi`	`fi`
`y :=x`	`y:=x` $\leftarrow$ *SEMANTIC*	`y:=a` $\leftarrow$ *TEXTUAL*
`output(y)`	`output(y)` $\leftarrow$ *SEMANTIC*	`output(y)`

The semantic and textual changes are determined by analyzing the program representation graphs for the two versions and computing a vertex correspondence between the two graphs.

Another research that addresses the limitation of the text-based tools is given in [40]. Yang has developed a tool that converts the two versions of C program into parse trees. Then it uses comparison algorithm based on dynamic programming to match the two trees. The differences are reported as deleted or updated nodes.

Chawathe et al. [5] have developed an algorithm that detects changes in structured data by calculating a "minimum-cost edit script" between the two data trees. The edit script consists of operations such as node insert, node delete, node update and subtree move. Using this script, one data tree can be converted into another.

One of the recent researches in this area is by Neamtiu et al. [31]. Their research is motivated by two factors. The first factor is dynamic software updating, and the second factor is providing the explanation of changes in software releases. Their approach is also tree based, comparing the abstract syntax tree of two versions of C programs. They track changes to global variables, types, and functions. The tool first finds the same named function in the two versions. It then compares the function bodies to compute a bijection between the types and variable names of the two program versions to determine changes. This tool also tries to detect renaming of names by checking that the new name does not exist in the old version and the old name does not exist in the new version. The changes are reported as the number of an artifact, such as function, global variable etc., present in version 1 and version 2 and the number of artifacts renamed. This tool has two limitations. First, it assumes that the function name does not change often, and second, it requires the ASTs to have the same shape.

The C Preprocessor (CPP) [13] provides an annotative approach for variation management. CPP is capable of expressing variations at a finer level and can handle multi-dimensional variations. However, this approach is unstructured and suffers from common pitfalls in macros such as incorrect

nesting, operator precedence, etc. Another major drawback of this approach is that the macros cannot be used to specify constraints between two or more macros.

One of the structured approaches of managing multi-dimensional variations is employed in Software product lines (SPLs) systems. SPLs [33] [34] facilitate an efficient and reusable method of creating variations of a program that is specific to a domain. A product is created by either adding or selecting the desired code fragments that encapsulate some functionality and are called features.

One of the approaches used for implementing SPLs is the so-called compositional approach. This approach utilizes the concept of the stepwise development paradigm that starts with a simple program and adds functionality incrementally, resulting in a complex program. Similarly, in SPLs, each program/product creation starts with a base code, which is then enhanced by using features that encapsulate certain functionality. The product is incrementally evolved by composing the current state of the product with the desired feature. AHEAD (Algebraic Hierarchical Equations for Application Design) [3], uses stepwise development and expresses the programs in terms of equations that are formed by constants and functions. The constants represent the base artifacts and the functions represent the refinements on the artifacts. An artifact can be created by the required function applications to the base artifact. If an artifact requires a chain of refinements, then it is modeled in AHEAD as series of function applications to the base artifact, which represents a constant. The customized product represented by the equations is then consumed by generators that synthesize the applications by composing feature refinements. The limitation of this approach is that it allows only coarse granularity of feature refinement because the refinements are either functions or modules. This approach cannot apply fine-grained refinements, such as change in function signature, to an artifact.

To some extent this limitation is addressed by tools, such as CIDE [19], which use the annotative approach. The annotative approach starts with a full-fledged product and then associates the pieces of code with a feature by using annotations. There may be an overlap between the code fragments

associated with more than one feature. CIDE annotates the programs with different colored features. The annotations are done on the abstract syntax tree of the language that restricts on the granularity level of the program that can be annotated. As a result, anything that is optional in the abstract syntax tree can be annotated while others, such as the return type, cannot be annotated. Although this tool provides a finer level of refinements than the compositional approach, still it restricts the granularity level of refinements based on the abstract syntax tree representation.

Other techniques, such as feature algebra, feature diagrams, propositional formulas based on feature models [2], are developed using either the compositional or the annotative approaches mentioned above. As a result, these suffer from similar limitations.

The limitations of the above-mentioned tools for handling multi-dimensional variations is addressed by the choice calculus [9] that uses the annotative approach. Among the other advantages that it offers, one of the advantage is that it allows the variations to be specified at the finest granularity level. Chapter 3 is devoted to variation management using the choice calculus.

2.3 Refactorings

Opdyke [32] addresses the problem of manually tracking and repeating the structural changes done in one part of the program to the other parts of the program resulting of naming, typing and scoping dependencies. As a solution, an automated process of restructuring is suggested that defines preconditions for each refactoring to ensure that program behavior is preserved. This research is focussed on supporting the iterative design of the object-oriented applications framework.

Fowler [11] defines refactoring as the process of changing a software system so that it does not alter the external behavior yet improves its internal structure. Refactoring improves the design of code that makes it easier to understand the code and to identify defects. The study done by Ratzinger et al. [35] confirms that refactorings can reduce the occurrence of defects in software.

The possibility of refactorings are identified by bad smells in the code. These bad smells can be attributed to duplicated code, long methods, lazy classes, etc. Depending on the bad smell, a set of refactorings can be chosen that helps in eradicating it. For example, duplicated code can be eliminated by extracting the method containing the duplicated code and replacing the duplicated code with a method call. Similarly, a long method can be decomposed by extracting a method. A long parameter list can be avoided by replacing the parameter with the method or by introducing a parameter object. A list of bad smells and the refactorings that can remove them are described by Fowler [11].

Below we discuss two examples that illustrate two refactorings, extract method and inline method.

Extract Method

The refactoring extract method can be used to eliminate several bad smells in code, such as duplication of code, long method etc. For example, the design of the following function `PrintOwing` can be improved by extracting the print statements in a different method.

```
void printOwing(double amount){
    printBanner();
    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

The restructured version of the above code is given as follows:

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}
void printDetails (double amount) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

The print statements are extracted in a new method `printDetails`. The extracted statements in the original function are replaced with a method call to the newly extracted function.

Inline Method

The refactoring inline method is used when a method body is concise and can be substituted for the method call. For example, the method `moreThanFiveLateDeliveries` returns the result of comparison of `_numberOfLateDeliveries` with `5`.

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

The method `moreThanFiveLateDeliveries` does not seem necessary and the method call to this can be replaced with the method body. The restructured code with the inlined method is given below:

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

A complete catalog of refactoring can be found in [11][10].

2.4 Identification of Refactorings

This section reviews the refactoring tools that compare the two program versions to identify the potential refactorings.

Demeyer et al. [7] provide a tool for understanding how the system has evolved by finding refactorings. This tool concentrates on three categories of refactorings, creation of template methods, incorporation of object composition relationships and optimisation of class hierarchies. This tool uses a heuristics-based approach to find refactorings and validates the applicability of these heuristics by testing on three case studies of object-oriented systems, including the VisualWorks, the HotDraw and the Refactoring Browser. To compare the results of three case studies, this tool

develops change metrics that can be derived from the object language source code. Some of the change metrics selected are lines of code in method body (Mthd-LOC), number of methods in the class (NOM), number of inherited method (NMI), etc.

In this approach, they first make certain research assumptions with respect to method size, class size and inheritance are used that help in determining the possibility of refactorings. One of the research assumption for method size is that the decrease in method size is a symptom of method split. These assumptions and change metrics are used to define heuristics that are recipes for identification of refactorings. Demeyer et al. describe the the following four heuristics:

1. Split into superclass
2. Split into subclass
3. Move to other class
4. Split method

This study claims to be reliable in identifying refactorings along with indicating the ordering of the parts based on likelihood of refactorings. However, this system is dependent on the names to anchor the pieces of code, which if changed can result in false positives. Another limitation is that some recipes require human intervention to check the source code of the two versions to deduce which refactorings have actually taken place. Lastly, the study mentions that reverse engineering based on heuristics require considerable resources. For cases where functions are renamed between two versions, S. Kim et al. [24] propose an automated algorithm that computes function similarity using the concept of origin relationship [14] explained below.

The approach extracts two consecutive versions $r1$ and $r2$, and computes a set of added and deleted functions as D and A. A function that exists in $r1$ and not in $r2$ is considered deleted and a function that exists in $r2$ and not in $r1$ is considered added. A candidate set is defined by the

multiplication of the two sets as $D \times A$. ‘‘A candidate set pair (d_x,a_y) has an origin relationship iff a_y is renamed and/or moved from d_x” [24].

After creating a candidate set, a set of eight similarity factors are computed for each function. These factors include, function name, signature, incoming call set, etc. For each pair in the candidate set, an overall similarity is computed by adding the values of similarity factors. If the overall similarity exceeds the threshold value, it indicates an origin relationship between the pair suggesting a function renaming of that pair between the two versions $r1$ and $r2$.

This research is limited to inferring only one composite change and it is dependent on the threshold value which is chosen incorrectly can lead to poor results.

Kim et al. [23] points out two limitations of the earlier refactoring tools. First, these cannot find the candidates that have more likelihood to be refactoring candidates. Second, the inferred changes are represented in an unstructured manner that may serve the purpose of reporting the changes. However, such representation of changes prevents further usage of the inferred change information by the software tools that could benefit from it.

They address these limitations by first, automatically finding the changes that are more likely to represent refactorings than the other potential matches. Next, by representing the inferred changes as first order relation logic rules that consists of scope, exception and transformation. For example, the following change rule specifies that all the methods whose class names either include `Plot` or `JThermometer` changes their package name from `chart` to `chart.plot`.

```
for all x in chart.* Plot*.*(*)
        or chart.*JThermometer*.*(*)
  packageReplace(x, chart, chart.plot)
```

The inference algorithm consists of the following four steps:

1. Generating seed matches.

2. Generating candidate rules based on seed matches.

3. Iteratively selecting the best rule among the candidate rules.

4. Post processing candidate rules to output set of changes.

The matching power of the inference algorithm is evaluated by finding the matching results for open source archives, JFreeChart, JHotDraw and JEdit. The results show that the inference algorithm can find more matches than the earlier techniques, such as [24].

Murphy-Hill et al. [29] have studied the existing refactoring-detection strategies to provide an experimental basis to the research assumptions related to refactorings that have been made in the past. Following are some of the interesting findings:

- Commit messages do not provide a complete picture of the refactorings. As a result, the tools that explore the commit logs for detecting refactorings such as [16] will leave out the refactorings.

- Large number of refactorings are low-level or medium level, so automatic refactoring detection tools that only identify high-level refactorings, such as modification of packages, classes etc., will miss out these refactorings.

- Programmers refactor frequently, this observation is also found in [39].

- Refactoring tools are under-used.

2.5 Identification of related changes from version differences

The tools discussed in this section aim to provide a better understanding of software evolution by analyzing the version differences to identify the the related changes. These tools intend to pro-

vide an efficient process of understanding the changes by reporting the changes from a high-level perspective.

Görg et al. [15] design a tool `REFVIS` that detects refactorings from software archives and visually represent the inferred refactorings using color coding and tool tips. This research infers structural refactorings, such as move class, move method etc., and local refactorings such as rename method, add remove parameter. The software archives are obtained from CVS and the information extracted from the repository is stored in a relational database using the approach mentioned in [42]. The following information is extracted form the repository:

- Version

  Represents one revision of a file, along with the information about the committer, the log message, timestamp, etc.

- Transaction

  Set of versions that have been committed at the same time by the same developer.

- Configuration

  A set of versions of distinct files.

Using the set of versions *V*, set of class names *C*, and set of methods *M*, two parsers are defined in [15] as follows:

- $parse_C : V \rightarrow P(C)$ returns the names of the classes contained in the version *V*.

- $parse_M : V \times C \rightarrow P(M)$ returns the signature of methods contained in class *C* in version *V*.

These parsers when applied to two consecutive versions *v* and *v'*, where *v'* is the predecessor version, can infer the following information about the class names:

- $ADDED_C(v,v') = parse_C(v) \setminus parse_C(v')$

- $REMOVED_C(v,v') = parse_C(v') \setminus parse_C(v)$
- $COMMON_C(v,v') = parse_C(v') \bigcap parse_C(v)$

The following information about common methods in the common class $c \in COMMON_c(v,v')$, can be obtained:

- $ADDED_M(v,v',c) = parse_M(v,c) \setminus parse_M(v',c)$
- $REMOVED_M(v,v',c) = parse_M(v',c) \setminus parse_M(v,c)$
- $COMMON_M(v,v',c) = parse_M(v',c) \bigcap parse_M(v,c)$

Additionally, other auxiliary functions such as $name_M(v,c,m)$, $parameter_M(v,c,m)$, are defined.

The refactorings are inferred by considering a version and its predecessor and defining conditions for parameters, return types etc. For example, the check for rename method is defined as follows in [15].

Rename Method

For every class $c \in COMMON_C(v,v')$ we consider all the method pair $(m_r,m_a) \in REMOVED_M(v,v',c)$ $\times$ $ADDED_M(v,v',c)$ that fulfill the following conditions:

$name_M(v',c,m_r) \neq name_M(v,c,m_a)$

$\wedge$ `parameters`$(v',c,m_r) =$ `parameters`(v,c,m_a)

$\wedge$ `returntype`$(v',c,m_r) =$ `returntype`(v,c,m_a)

$\wedge$ `body`$(v',c,m_r) =$ `body`(v,c,m_a)

The pairs (m_r,m_a) describe a Rename Method, with the name $name_M(v',c,m_r)$ in class c renamed as $name_M(v,c,m_a)$. The checks for other refactorings are described in a similar manner.

After checking the existing two versions for refactorings, the algorithm picks the next predecessor version and the whole process is repeated.

Tools such as CatchUp and JBuilder use record-replay of refactorings to automatically update the applications with the refactorings that are done to a single component. However, refactorings done by using refactoring logs can lead to defects as the logs lack information about manual refactorings [8]. As a solution, Dig et al. introduces an algorithm that does a syntactic analysis on the parse trees of the two versions and uses the Shingles encoding [4] to find the similar pair of entities called refactoring candidates. Next, the algorithm does a semantic analysis using refactoring detection strategies to ensure if these candidate represent refactorings. For example, if the candidates are two similar methods, then the semantic analysis checks if the method name, parameter list and method body are same. If the last two are same, then it reports that the method has been renamed.

The semantic analysis uses the following seven refactoring detection strategies:

1. RenamePackage (RP)
2. RenameClass (RC)
3. RenameMethod (RM)
4. PullUpMethod (PUM)
5. PushDownMethod (PDM)
6. MoveMethod (MM)
7. ChangeMethodSignature (CMS)

A refactoring log (`rlog`) is used that records the refactorings that are detected. At any given time, a strategy compares the two candidates and uses the information about the refactorings detected till that time. Each strategy specifies a list of syntactic and semantic checks. The syntactic checks are done using the function ρ, with g_1 and g_2 for the graphs for the two versions.

The function ρ defined in [8] recursively checks if any part of the fully qualified name has been renamed. It uses two functions `pre` and `suf` for prefix and suffix, and `fqn` is the fully qualified name.

$$\begin{aligned}\rho(\texttt{fqn},\texttt{rlog}) = \;& \texttt{if}\ (\texttt{defined}\ \texttt{rlog}(\texttt{fqn}))\ \texttt{then}\ \texttt{rlog}(\texttt{fqn}) \\ & \texttt{else}\ \rho(\texttt{pre}(\texttt{fqn}),\texttt{rlog})\ +\ \texttt{"."}\ +\ \texttt{suf}(\texttt{fqn}) \\ \rho(\texttt{""},\texttt{rlog}) = \;& \texttt{""}\end{aligned}$$

For example, a rename strategy `RM`(m_1,m_2), uses the function ρ to find whether the fully qualified name of a candidate from the pair, matches the renaming in the `rlog`. The strategy for rename method shows three syntactic checks:

$$m_2 \notin \texttt{g1}$$
$$\rho(\texttt{pre}(m_1),\texttt{rlog}) = \texttt{pre}(m_2)$$
$$\texttt{suf}(m_1) \neq \texttt{suf}(m_2)$$

The first check specifies that m_2 should not exist in g_1. The second check ensures that the prefix of the fully qualified name of both the methods are same. This ensures that the method is renamed and not moved. The third check ensures that the name of the methods are different.

The semantic check is based on the directed similarity between the two nodes n and n'. The directed similarity is computed by finding the ratio of the similar incoming edges to both the nodes and the total number of edges. The similar incoming edges are found, by first finding the node n_i with incoming edge to node n and then finding a node n_i' that corresponds node n_i using the information in the `rlog`. Then an overlap between the edges from n_i to n and n_i' to n' is computed, and is divided by the total number of incoming edges. In a similar manner, another directed similarity is computed by considering the node n' first. An overall similarity (σ) is computed by taking an

average of both the directed similarities.

The semantic check for rename strategy $\texttt{RM}(m_1,m_2)$, is defined as:

$$\sigma(m_1,m_2,rlog) \geq \texttt{T}$$

If the overall similarity exceeds the threshold value T, then the semantic check for the above strategy is satisfied. The implementation of this algorithm is available as RefactoringCrawler.

Weißgerber et al. [38] have proposed a system that detects refactorings from the source-code changes and ranks these as low or high risk refactorings. The system pre-processes the version data and stores the information in relational database. Next, it collects a set of versions that have been committed by the same developer at the same time and group these as transactions. The refactorings candidates are then selected by doing a syntactic and structural analysis of the transactions. A candidate consists of an entity with the associated version, and the successive version of entity along with the version. Using the equality testing and clone-detection algorithm, these candidates are categorized as equal, clone or no clone. The candidates belonging to equal and clone set form the low risk refactorings, whereas the no clone refactorings are high risk refactorings.

In contrast to the above approaches that use the existing repositories to detect refactorings, Robbes [36] describes the idea of change-based repository. Instead of containing the delta representations that represent the differences between the two versions, the change-based repository consists of set of change operations that have been performed on the programs. These change operations are collected by monitoring the interaction of the programmer with the IDE while developing the software.

In the proposed repository the programs are represented as ASTs with change history associated with each entity. The two types of change operations that are stored are, atomic, representing the low-level changes and composite, representing the high-level changes, such as refactorings.

The change-based history is stored as trees with root representing one complete change, leaves representing the low-level changes, and the intermediate levels representing the composite changes. This system is implemented as the plug-in SpyWare in Squeak Smalltalk environment. The main disadvantage of this system is that it relies on the IDE interactions that prevents it from capturing the refactorings performed manually.

Kim et al. [22] address the limitation of program differencing tools that report differences as individual changes, even when the changes are related. This research designs a Logical Structural Diff tool (LSDiff) that reports the low-level changes that are related as high-level changes.

The two program versions P_o and P_n are represented by a fact-based representation as FB_o and FB_n. In the fact-based representation a program version is represented as a set of predicates such as:

```
package(packageFullName)
method(methodFullName, methodShortName, typeFullName)
```

These predicates describe the code elements, their containment relationships and their structural dependencies. For example, `method("BMW.start","start","BMW")` in a fact-base indicates that the corresponding program version consists of class `BMW` and implements `start` function. A fact-level difference (ΔFB) is computed by taking the set difference between the FB_o and FB_n. For example, if FB_n consists of a fact `calls("BMW.start","Key.chk")` that does not exist in FB_o, then ΔFB will have an entry `+calls("BMW.start","Key.chk")`, where `+` indicates an added fact. Next, the facts in the three fact-bases are distinguished by adding the prefix `past_`,`current_` to FB_o and FB_n respectively, and `added_` and `deleted_` to ΔFB. The facts in three fact-bases are then used to infer logic rules that represent systematic changes. Following are the three rule styles:

- `past_`$* \Rightarrow$ `deleted_`$*$

This rule indicates a feature removal. For example, the following indicates that all the methods that called the method `DB.exec` in the old version deleted the calls to `DB.exec`.

`past_calls(m,"DB.exec")` ⇒ `deleted_calls(m,"DB.exec")`

- `past_*` ⇒ `added_*`

  This rule states that codes in the previous versions with the similar characteristics added the similar code. For example, the following rule indicates that all the methods in the previous version that accessed `Log.on` added called to `Log.trace`.

  `past_accesses("Log.on",m)` ⇒ `added_calls(m,"Log.trace")`

- `current_*` ⇒ `added_*`

  This rules states a feature addition. For example, the following rule implies that all the subtypes of `Svc` added the new method `getHost`.

  `current_method(m,"getHost",t)` ∧ `current_subtype("Svc",t)`
  ⇒ `added_method(m,"getHost",t)`

- `deleted_*` ⇒ `added_*` or `added_*` ⇒ `deleted_*`

  This rule states the dependencies between the additions and deletions. For example, the following rule states that all entities that deleted the method `getHost`, inherited the `getHost` field from `Service`.

  `deleted_method(m,"getHost",t)`
  ⇒ `added_inheritedfield("getHost","Service",t)`

A fact f in Δ`FB` matches a rule r if f can be obtained by replacing in the consequent the values that satisfy the antecedent of the rule. For example, suppose the following rule is inferred:

`past_accesses("Key.on",m)` ⇒ `deleted_accesses("Key.on",m)`

If the factbase $\texttt{FB}_n$ consists of the following three facts:

```
-accesses("Key.on","GM.start")
-accesses("Key.on","Kia.start")
-accesses("Key.on","Bus.start")
```

These three facts when compared with the above-mentioned inferred rule result in a new version of $\Delta\texttt{FB}$, say $\Delta\texttt{FB}'$ with the three facts replaced with the rule that these match to. Next, the rest of the facts in the resulting factbase is compared with the inferred rules. In addition to detecting the changes, this research also identifies anomalies to the detected changes. An anomaly occurs if no match for the rule is found in the $\Delta\texttt{FB}$ or its updated versions.

Kim et al. [21] have recently developed the plug-in Ref-Finder for Eclipse that visually presents the identified refactorings instances. This plug-in uses a template-based approach for detection of refactorings. In this approach, first the structural constraints in terms of template logic queries are described, along with the ordering dependencies among the different refactorings. The plug-in takes two versions of a program and using the AST analysis of Eclipse extracts logic facts related to the syntactic structure of the program. Next, it uses the Tyruba (Type Rule Base) logic programming engine [37] to match the identified program changes with the constraints related to each refactorings pre-defined as template logic queries.

2.6 Differences with our Approach

Previous sections have discussed some of the existing version management tools and the refactoring detection tools. The first category of refactorings tools that we have discussed infer refactorings by analyzing the two program versions and match these with the pre-defined checks. The second category of refactorings detection tools either exploit existing software repositories or create their own repository containing the low-level change information. In this section, we draw a comparison

between the existing research work and the system proposed in this thesis.

It is necessary to mention here that this thesis does not intend to create a version management tool. Instead, it points out the limitations of the existing version differencing tools and how these can be used to build a system that can provide better explanation of program changes. The existing version differencing tools [30] [18] [5], either cannot find the structural differences between the two program versions, or report the differences as atomic changes. The differences when reported as atomic changes can result into a time consuming process of understanding program differences. Additionally, these tools represent the changes in such as manner that the changes can describe the differences, but cannot be used by other software tools that may want to explore these changes further.

This thesis addresses the first limitation by reporting the differences at a higher abstraction level. The second limitation is addressed by using a formal representation for the changes detected by the version differencing tools. The annotative and the compositional variation management tools such as CPP, CIDE, AHEAD cannot represent the fine-grained variations. However, the choice calculus overcomes this limitation and provides a modular, flexible and in-place representation of variations. Hence, we use choice calculus to represent the variations in this thesis.

The tools [7] [24] compare the two program versions by matching the program elements and further comparing these against predefined conditions to detect refactorings. These tools are limited in the number of refactorings that these can detect. Additionally, these either require human intervention or require considerable resources.

Although [15] present the inferred changes using the visualizations such as color-codings etc., it can only infer refactorings that are limited to class structure or methods. On the other hand, Dig et al. [8] cannot find the refactorings one to class fields and interface methods and the refactoring detection is dependent on the threshold value which if selected too high gives poor results. Additionally, some of the researches, such as Weißgerber [38] cannot detect multiple refactorings done to the same

expression.

In comparison to the previously discussed researches, our approach can infer several refactorings including multiple refactorings done to the same expression, without requiring considerable resource. In addition to the differences discussed above, none of the above-mentioned refactoring approaches point out the existing exception cases(if any) in the code. These exception cases help in identification of possible defects in the code.

The research work done by Kim et al. [22] shares the same goals as our work, yet our approach differs from theirs in many respects. Kim [22] represent the program versions as fact-bases and compares the fact-bases of the old and the new program version to create the delta fact-base that represents the difference between the two versions. Representing the entity information as facts can result in loss of contextual information in the fact-bases. However, in our system the differences are represented using choice calculus that allows differences to be described in-place with no loss of contextual information.

Since the fact-bases may not contain the entire context information, the differences as well as the original fact-bases are analyzed to create logic rules. Instead, in our approach, the variation expression representing the program differences is compared with the variation pattern corresponding to the refactorings. The variation pattern for a refactoring describes the expected structure of the variation expression for that refactoring.

Additionally, in this thesis the refactorings are selected on the basis of the structure of the variation expression. Our system annotates the inferred refactorings in the variation expression, which allows these composite changes to be explored for further analysis.

Chapter 3 – The Choice Calculus

3.1 Introduction

The problem of software variation management is multi-faceted. The two most common areas of variations in software are variations with respect to time and variations with respect to domain. The first area of variation i.e. temporal variation has been a vast field of study and has resulted in the development of various tools under the category of revision control systems and software configuration management systems. The second area of variation i.e. non-temporal variation has been handled by software configuration management systems, C preprocessor (CPP), and software product lines (SPLs).

The different areas of variation management research do not employ a common approach to deal with variations. As a result, there is a lack of a formal basis for variation management research. The absence of a common foundation for software variation management research in different areas of variation has been addressed by development of the choice calculus [9]. The choice calculus provides a representation for software variations and serves as a theoretical foundation for variation management research.

In our research to design a system that provides an efficient reporting of software changes, choice calculus plays a pivotal role. The choice calculus is the variation representation that is used in this thesis to provide a structured representation to the atomic changes reported by a version differencing tool. These variations represented by choice calculus serve as input to the composite change inference algorithms that analyze these variations to infer composite changes. Therefore, it is essential to discuss the design of the choice calculus and the various features that it offers.

This chapter describes the choice calculus and the associated mechanics behind it. The existing tools that deal with different areas of software variation management are discussed in Section 2.2. Section 3.2 discusses the variation representation using the choice calculus including its syntax and semantics. This section is important as in the latter chapters of this thesis we use the choice calculus extensively to represent variations.

3.2 The Choice Calculus

To understand the variation representation offered by the choice calculus, let us reconsider the example presented in the paper [9]. The following two expressions, namely e_1 and e_2, are the two implementations of the function `twice` that takes a parameter and returns the parameter value doubled. In both the expressions, the function `twice` is implemented using a plus operator (+) as follows:

$$e_1 \equiv \texttt{twice x = x+x}$$

$$e_2 \equiv \texttt{twice y = y+y}$$

The above two implementations of the function `twice` are similar; they differ only with respect to the name of the parameter. The first implementation has parameter name as `x` and the second as `y`. The domain of variation such as parameter name in the above example, is called a *dimension of variation* in the paper [9].

The implementation in e_1 and e_2, doubles a value using the plus operator (+). As the values can also be doubled by multiplying it with `2`, another dimension of variation i.e. implementation method can be added. This new dimension of variation adds two more variants, e_3 and e_4 of the

$$
\begin{array}{lcll}
d & ::= & A \mid B \mid \ldots & \textit{Dimension Name} \\
t & ::= & a \mid b \mid \ldots & \textit{Tag Name} \\
v & ::= & \underline{a} \mid \underline{b} \mid \ldots & \textit{Reference Variable} \\
e & ::= & a\text{-}\prec e,\ldots,e\succ\text{-} & \textit{Structure} \\
 & \mid & \textbf{dim}\ d\langle t,\ldots,t\rangle\ \textbf{in}\ e & \textit{Dimension} \\
 & \mid & d\langle e,\ldots,e\rangle & \textit{Choice} \\
 & \mid & \textbf{let}\ v{=}e\ \textbf{in}\ e & \textit{Binding} \\
 & \mid & v & \textit{Reference Variable}
\end{array}
$$

Figure 3.1: Choice Calculus

function `twice` as follows:

$$e_3 \equiv \texttt{twice x = 2*x}$$

$$e_4 \equiv \texttt{twice y = 2*y}$$

A variation representation should be able to represent the above-mentioned four variants, along with the respective dimensions of variations. Additionally, it should provide the mechanism to select any of the possible variants. The subsections that follow discuss the representation of the variants using choice calculus and the methodology provided by the choice calculus to manage these variations.

3.2.1 Syntax

The syntax of the choice calculus is described in Figure 3.1. The construct *structure* represents the symbols that are varied and are stored in choice calculus constructs. For representing variations in a given object language, the structure construct is replaced with the object language constructs.

To discuss the other constructs, let us reconsider the two variants e_1 and e_2 of the function `twice` that vary with respect to parameter name. Using the syntax of the choice calculus shown in the Figure 3.1, the variations between the expressions e_1 and e_2 are described as follows:

$$e_p \equiv \textbf{dim}\ Par\langle x,y\rangle\ \textbf{in}\ \texttt{twice}\ Par\langle \texttt{x},\texttt{y}\rangle = Par\langle \texttt{x},\texttt{y}\rangle + Par\langle \texttt{x},\texttt{y}\rangle$$

Figure 3.1 shows that a *dimension* consists of a dimension name (*d*), set of tags *(t,...,t)* and the dimension expression *e* that represents the varied expression. A *dimension name* is a string that specifies the dimension of a variation. The *set of tags* indicate the number of alternatives that exist in the variation where each tag corresponds to one alternative of the variation. The dimension expression *e* that consist of variations with respect to the related dimension. The expression e_p is a dimension with dimension name *Par*, the set of tags $\{x,y\}$ indicate two alternatives for the variation, and the expression that is varied is $\texttt{twice}\ Par\langle \texttt{x},\texttt{y}\rangle = Par\langle \texttt{x},\texttt{y}\rangle + Par\langle \texttt{x},\texttt{y}\rangle$.

A *choice* is a part of a dimension and shares its dimension name. It consists of the set of alternatives that exist for the dimension it is related to. For example, the update of `x` to `y` in the expressions e_1 and e_2 is defined as a choice $Par\langle \texttt{x},\texttt{y}\rangle$. The dimension name *Par* denotes the dimension that the choice corresponds to, and the two values `x` and `y` denote the two alternatives for the dimension *Par*. The choice calculus allows the choices to be described at the point of variation in the expression.

The set of alternatives in a choice correspond to the set of tags in the dimension by their position. This implies that the number of tags specified in a dimension should be same as the number of alternatives in the choices that are in the dimension's scope. A dimension expression that follows this property is called a *well-dimensioned* expression.

The association of tags and alternatives in a dimension allows the selection of the variants from a dimension. This requires a tag selection that first finds the next dimension declaration and selects a qualified tag from the dimension. A *qualified tag D.t* specifies an existing dimension with the

dimension name D and an existing tag name t. The next step is *choice elimination* that eliminates choices that are bound to the selected dimension and replaces each choice with the alternative that corresponds to the selected tag. Given an expression **dim** $D\langle t_1,\ldots,t_n\rangle$ **in** e with the selection of qualified tag $D.t_i$, the choice elimination formally described as $\lfloor e\rfloor_{D.i}$ replaces each choice of the form $D\langle e_1,\ldots,e_n\rangle$ with its i^{th} alternative e_i. After all the choices related to the selected dimension are replaced, the dimension declaration is eliminated. The formal definition of choice elimination is defined as follows:

$$\lfloor v\rfloor_{D.i} = v$$

$$\lfloor a\prec e_1,\ldots,e_n\succ\rfloor_{D.i} = a\prec\lfloor e_1\rfloor_{D.i},\ldots,\lfloor e_n\rfloor_{D.i}\succ$$

$$\lfloor \textbf{let } v{=}e \textbf{ in } e'\rfloor_{D.i} = \textbf{let } v{=}\lfloor e\rfloor_{D.i} \textbf{ in } \lfloor e'\rfloor_{D.i}$$

$$\lfloor \textbf{dim } D'\langle t^n\rangle \textbf{ in } e\rfloor_{D.i} = \begin{cases} \textbf{dim } D'\langle t^n\rangle \textbf{ in } e & \text{if } D = D' \\ \textbf{dim } D'\langle t^n\rangle \textbf{ in } \lfloor e\rfloor_{D.i} & \text{otherwise} \end{cases}$$

$$\lfloor D'\langle e_1,\ldots,e_n\rangle\rfloor_{D.i} = \begin{cases} \lfloor e_i\rfloor_{D.i} & \text{if } D = D' \\ D'\langle\lfloor e_1\rfloor_{D.i},\ldots,\lfloor e_n\rfloor_{D.i}\rangle & \text{otherwise} \end{cases}$$

A variant selection from expression e_p requires a selection of a qualified tag *Par.x* or *Par.y*. The selection of *Par.x* will select the first alternative from all the choices, resulting in expression e_1. Similarly, the tag *Par.y* will result in selecting expression e_2. The variants e_1 and e_2 are called *variation-free* due to the absence of variations in the form of choices or dimensions.

As the expression e_p consists of a single dimension with all the choices in the scope of the same dimension, the tag selection will select the same alternative in all the choices. Such a specification can be used whenever it is required that the same alternative has to be selected. On the other hand,

all the choices may not be in the scope of a single dimension as shown in the following expression:

$$e'_p \equiv \texttt{twice}\ \textbf{dim}\ Par\langle x,y\rangle\ \textbf{in}\ Par\langle \texttt{x},\texttt{y}\rangle\ = \\ \textbf{dim}\ Par\langle x,y\rangle\ \textbf{in}\ Par\langle \texttt{x},\texttt{y}\rangle + \textbf{dim}\ Par\langle x,y\rangle\ \textbf{in}\ Par\langle \texttt{x},\texttt{y}\rangle$$

In this expression all the choices are independent which means that each dimension can select either of the alternative from the choices. The following expression is one of the variant of the expression e'_p:

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

Therefore, for the situations when all the variants are related to the same dimension of variation, a single dimension is defined containing all the related choices. A limitation of having more than one occurrence of same choice is that it can result into update anomalies. These anomalies can be avoided by using a *binding* construct. A binding construct is a let expression that consists of a reference variation, a definition of the reference variable, and the scope that consists of occurrences of the reference variable. For example, expression e''_p can be defined as:

$$e''_p \equiv\ \textbf{let}\ \underline{v}=\textbf{dim}\ Par\langle x,y\rangle\ \textbf{in}\ Par\langle \texttt{x},\texttt{y}\rangle\ \textbf{in}\ \texttt{twice}\ \underline{v} = \underline{v}+\underline{v}$$

This expression ensures that the alternative that is selected by the tag selection in dimension *Par* is substituted for all the occurrences of v and avoids any update anomalies. A binding construct consists of a reference (v), an expression that is assigned to the reference and the expression in which the reference is to be substituted with the value that is assigned to it. In such expressions, the selection is done once and the same value is used at all the occurrences of v. In a given expression, if the bindings are reduced i.e. reference is substituted in the expression, then the let expression gets

reduced and the expression becomes *sharing-free*.

An expression such as $e"_p$ can be converted into the following variation-free expression by selecting the tag *Par.x*:

$$\textbf{let } \underline{v}{=}\texttt{x} \textbf{ in } \texttt{twice } \underline{v} \;=\; \underline{v}{+}\underline{v}$$

On substituting the value of reference in the expression, the binding construct can be eliminated resulting into a sharing-free expression as follows:

$$\texttt{twice } x \;=\; \texttt{x+x}$$

The above expression that is both variation-free i.e. no choices or dimensions and sharing-free i.e. no let constructs, and is called a *plain* expression.

In the later sections, the string "**let**" in the binding construct is replaced with "**share**" in order to differentiate it from the let binding in lambda calculus.

Let us now consider the second dimension of variation i.e. implementation method that adds the two variants e_3 and e_4. A new dimension with dimension name *Impl* is added to the expression e_p resulting into the following expression:

$$e_i \;\equiv\; \textbf{dim } \mathit{Impl}\langle \mathit{plus}, \mathit{times}\rangle \textbf{ in } \mathit{Impl}\langle e_p, e_t\rangle$$

where

$$e_t \;\equiv\; \textbf{dim } \mathit{Par}\langle x, y\rangle \textbf{ in } \texttt{twice } \mathit{Par}\langle \texttt{x}, \texttt{y}\rangle \;=\; \mathit{Par}\langle \texttt{x}, \texttt{y}\rangle * \mathit{Par}\langle \texttt{x}, \texttt{y}\rangle$$

The expression e_i consists of two dimensions, one with dimension name *Impl* and second with

dimension name *Par*. The expression e_i is well-dimensioned as the number of tags specified in both the dimensions is same as the number of alternatives in the choices in its scope. An alternate way of describing the expression e_i is by factoring out the dimension *Par* as follows:

$$e'_i \equiv \textbf{dim}\ Impl\langle plus, times\rangle\ \textbf{in dim}\ Par\langle x,y\rangle\ \textbf{in}\ \texttt{twice}\ Par\langle \texttt{x},\texttt{y}\rangle\ =$$
$$Impl\langle Par\langle \texttt{x},\texttt{y}\rangle + Par\langle \texttt{x},\texttt{y}\rangle, Par\langle \texttt{x},\texttt{y}\rangle * Par\langle \texttt{x},\texttt{y}\rangle\rangle$$

The nesting of one dimension in the other suggests the dependence of the dimensions. With nested dimensions, a variant is obtained by making the selections from the outermost dimension to the inner most dimension. The number of tag selections are dependent on the number of dimensions in an expression. To select a variant from e_i, first *Impl.plus* is selected that results in the following expression:

$$\textbf{dim}\ Par\langle x,y\rangle\ \textbf{in}\ \texttt{twice}\ Par\langle \texttt{x},\texttt{y}\rangle\ =\ Par\langle \texttt{x},\texttt{y}\rangle + Par\langle \texttt{x},\texttt{y}\rangle$$

Next, the tag *Par.y* is selected that results into the following variation-free expression:

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

3.2.2 Choice Semantics

The semantics of a choice expression is a mapping from tuples of tags to plain expressions. A tuple of tags include the different qualified tags that have to be selected from a given choice calculus expression to obtain a plain expression. The choice semantics is a two step process. The first step involves defining a set of mappings $V(e)$ between tuples of tags (q) to variation-free expressions.

The second step includes expansion of all the let constructs in the variation-free expression obtained from first step, to generate a plain expression.

For example, the expression e_i in the previous section, consists of two dimensions *Impl* and *Par*. To obtain a variation-free expression, the tag selection from both the dimensions is required. As each dimension contains two tags, a total of four tag selection combinations are possible. These four possible combinations of tag selections and the corresponding variants are given in the following set:

$$
\begin{aligned}
V(e_i) = \{ & \\
& ((\textit{Impl.plus}, \textit{Par.x}), \texttt{twice x = x+x}), \\
& ((\textit{Impl.plus}, \textit{Par.y}), \texttt{twice y = y+y}), \\
& ((\textit{Impl.times}, \textit{Par.x}), \texttt{twice x = x*x}), \\
& ((\textit{Impl.times}, \textit{Par.y}), \texttt{twice y = y*y}) \\
\} &
\end{aligned}
$$

As the variation-free expression do not contain let expressions, the second step of expanding the let expressions is not needed. Hence, the variation-free expressions given above are the plain expressions that can be obtained from the expression e_i.

Chapter 4 – Combining Object Language and Variation Representation

4.1 Introduction

Object language programs go through changes that generate different versions of the same program. The differences between any two versions should be reported in such a way that the changes are identified and understood in an efficient manner. As discussed in chapter 1, the changes described in a structured manner allows the changes to be further analyzed. Additionally, changes when reported as composite changes provide a high-level view of changes making it easier and time efficient for the programmer to understand the changes.

The structure to the changes is provided using a *variation representation* which allows the change inference algorithms to analyze these changes. Representing the changes in an object language requires enhancing the object language with variation representation capabilities. One straightforward way of achieving such capabilities in an object language is to combine it object with an appropriate variation representation, to design a new language called *variational object language*.

Various factors influence the design of a variational object language. These factors can be general language design principles, such as conciseness, non-redundancy etc., or these can be specific to the object language, such as what terms need variations, how algorithms interact with the terms, etc. These factors result in more than one approach for designing a variational object language, with each approach offering a different representation. A particular approach for a representation of a variational object language is selected by determining the factors that influence the design the most. It is necessary to study the available approaches for designing the representation, so that the best

possible representation can be selected.

This chapter discusses the different approaches that can be used in designing a variational object language. The conciseness and completeness lambda calculus makes it a good choice as an object language. As discussed in chapter 3, choice calculus provides a general and structured representation for multi-dimensional variations in a software. The advantages, such as flexibility and structure resulted in the selection of choice calculus as variation representation. Therefore, the approaches discussed in the following sections use *lambda calculus* as object language and the *choice calculus* as variation representation. The language resulting from combination of lambda calculus with choice calculus is called as *variational lambda calculus (VLC)*.

4.1.1 Example

Consider the following lambda calculus expression e_{old} that defines an abstraction as follows:

$$e_{old} \equiv \lambda \texttt{x.succ x}$$

Suppose, later the expression e_{old} is changed to the following expression e_{new}:

$$e_{new} \equiv \lambda \texttt{x.pred x}$$

Expressions such as e_{old} and e_{new} are called *plain expressions*, where the term *plain* refers to the absence of variations. To represent the changes between the two lambda calculus expression e_{old} and e_{new} using choice calculus, the language variational lambda calculus is designed. This new language includes the constructs from lambda calculus and choice calculus, so that it can represent

the differences between e_{old} and e_{new} as follows:

$$e_D \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \lambda \mathtt{x}.D\langle \mathtt{succ}, \mathtt{pred}\rangle\ \mathtt{x}$$

An expression similar to e_D is called *variation expression* as it contains variations in the form of the choice calculus constructs. The variation expression e_D is a dimension with dimension name D, two set of tags *old* and *new*, and the dimension expression $\lambda \mathtt{x}.D\langle \mathtt{succ}, \mathtt{pred}\rangle\ \mathtt{x}$. The *dimension* D denotes the dimension of variation which in the above example is time. The two tags suggest the two alternatives for the variation. The dimension expression is an abstraction with the choice $D\langle \mathtt{succ}, \mathtt{pred}\rangle$ containing two alternatives where `succ` corresponds to the tag *old* and `pred` corresponds to the tag *new*. Either of the alternatives can be selected from the choice by using choice elimination described in page 34. The selection of qualified tag *D.old* will select the expression e_{old} and the selection of tag *D.new* will select e_{new}.

The following sections study the different approaches that are available for designing variational lambda calculus. The new language should be able to represent changes as described in the expression e_D and should support the selection of variants as described in the example above. The design starts with the discussion of a basic approach for variational lambda calculus in Section 4.2. Section 4.3 presents the *parameterized choice calculus* approach that extends the range of variations that can be specified and allows variations to be specified selectively. Section 4.4 extends the syntax to make it scalable to deal with languages with large number of constructs. Section 4.5 discusses a trade-off between two important ways of dealing with illegal expressions that can created by the syntax. And finally Section 4.7 presents the chosen representation for variational lambda calculus and the corresponding Haskell syntax.

4.2 Basic Approach

x	::=	`a` | `b` | ...	*Variable*
e	::=	x	*Variable*
	|	$\lambda x.e$	*Abstraction*
	|	$e\ e$	*Application*
	|	`let` x=e `in` e	*Let Binding*

Figure 4.1: Lambda Calculus

d	::=	$A \mid B \mid \ldots$	*Dimension Name*
t	::=	$a \mid b \mid \ldots$	*Tag Name*
v	::=	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
CC	::=	$a\prec CC,\ldots,CC\succ$	*Structure*
	|	**dim** $d\langle t,\ldots,t\rangle$ **in** CC	*Dimension*
	|	$d\langle CC,\ldots,CC\rangle$	*Choice*
	|	**let** v=CC **in** CC	*Binding*
	|	v	*Reference Variable*

Figure 4.2: Choice Calculus

Figure 4.1 shows the syntax of lambda calculus with four constructs - variable, abstraction, application and let binding. Chapter 3 discusses the syntax of choice calculus consisting of structure, dimension, choice binding and reference constructs shown in Figure 4.2. We have replaced e with CC in order to differentiate it with the lambda calculus terms.

The first approach that we use for introducing variations in an object language using choice calculus is the same as mentioned in the paper [9]. This approach suggests replacing the structure construct of choice calculus with all the lambda calculus constructs in order to represent variations in lambda calculus.

d	::=	$A \mid B \mid \ldots$	*Dimension Name*
t	::=	$a \mid b \mid \ldots$	*Tag Name*
v	::=	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
x	::=	`a` $\mid$ `b` $\mid \ldots$	*Variable*
e	::=	x	*Variable*
	$\mid$	$\lambda x.e$	*Abstraction*
	$\mid$	$e\ e$	*Application*
	$\mid$	`let` x=e `in` e	*Let Binding*
	$\mid$	**dim** $d\langle t,\ldots,t\rangle$ **in** e	*Dimension*
	$\mid$	$d\langle e,\ldots,e\rangle$	*Choice*
	$\mid$	**let** v=e **in** e	*Binding*
	$\mid$	v	*Reference Variable*

Figure 4.3: Variation Lambda Calculus-Version 1

Using the basic approach, the resulting syntax of variational lambda calculus is shown in Figure 4.3. As e ranges over lambda calculus and choice calculus constructs, while describing an expression e can be substituted with constructs from either of the languages. As a result, lambda calculus expressions with nested choice calculus expressions and vice versa can be specified in variational lambda calculus. Using the given syntax, the variation expression e_D discussed in Section 4.1.1 can be described in variational lambda calculus.

Chapter 3 discussed the process of *choice elimination* to select a variant from a variation expression. As a result, for the variation expression e_D, the qualified tag *D.old* selects the variant λ`x.succ x` and the tag *D.new* selects λ`x.pred x`.

Advantages:

The advantages of this approach are described as follows:

1. *Simple and Easy*

This basic approach of introducing variations in an object language is fairly intuitive. To create variational lambda calculus, it simply requires replacing the structure construct of choice calculus with the lambda calculus constructs.

Limitations:

The limitations of this approach are described as follows:

1. *Requires creation of a new language and functionality*

   Although the approach discussed above is simple and intuitive, it becomes tedious and complex when dealing with the functionality such as semantics, associated with the two languages. As the new language represented by the term *e* is created, the functionality associated with lambda calculus and choice calculus cannot be reused. As a result, the associated functionality with the two languages has to be rewritten to handle the constructs of the new language.

2. *Unmanageable and Inconsistent functionality*

   Consider the object language with more than one non-terminal in Figure 4.4. With the above-

$$
\begin{array}{lcl}
\textit{val} & ::= & \texttt{1} \mid \texttt{2} \mid \ldots \\
\textit{var} & ::= & \texttt{a} \mid \texttt{b} \mid \ldots \\
e & ::= & \textit{add } e\ e \\
 & \mid & \textit{sub } e\ e \\
 & \mid & \textit{gt } e\ e \\
 & \mid & \textit{lt } e\ e \\
 & \mid & \textit{val} \\
 & \mid & \textit{var} \\
\textit{stmt} & ::= & e \\
 & \mid & \textit{assign var stmt} \\
\textit{loop} & ::= & \textit{while stmt stmt} \\
 & \mid & \textit{for stmt bop bop stmt}
\end{array}
$$

Figure 4.4: Example Object Language

mentioned approach, a new instance of choice calculus will be created for each of the non-terminals mentioned above. And each new instance of choice calculus implies rewriting of the functionality for choice calculus that accommodates the corresponding object language non-terminal. This will result in creation of multiple copies of the choice calculus functionality that may not be consistent and can lead to update anomalies when the functionality needs to be changed. For example, changing the semantics of one of the choice calculus construct will require the same change in each of the instance of choice calculus.

Similarly, any change in any construct of the object language would require changes in the choice calculus instance consisting of the changed construct. For example, if a new construct is added to the non-terminal *stmt*, then besides adding the object language functionality, it will require the choice calculus functionality to be extended to handle the new construct. Similarly, any modifications to any of these construct will also require changes in the functionality of choice calculus.

This illustrates that if the object language and the variation representation are so closely tied, making modifications in the functionality of either requires modification in the other language. Specifically, for object language with more than one non-terminals, re-instantiating the choice calculus for each non-terminal can lead to inconsistent and unmanageable code.

3. *Limited Variations*

   The syntax given in Figure 4.3 is limited in terms of variations that it can represent which is demonstrated by the example that follows. Suppose the expression e'_{old} is changed to the expression e'_{new} as defined below:

$$e'_{old} \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 2}$$

$$e'_{new} \equiv \texttt{let g=}\lambda\texttt{x.succ x in g 2}$$

The expression e'_{old} defines a function `f` as an abstraction that applies the function `succ` to the argument, and in the scope of the let expression the function `f` is applied to a value `2`. The difference between e'_{old} and e'_{new} is that the variable `f` is replaced with the variable `g` in the function name as well as the scope. Hence, the variation expression should represent the difference between e'_{old} and e'_{new}, using a choice expression $D\langle \texttt{f},\texttt{g}\rangle$ in the function name and the scope as follows:

$$e'_D \equiv \texttt{let } D\langle \texttt{f},\texttt{g}\rangle\texttt{=}\lambda\texttt{x.succ x in } D\langle \texttt{f},\texttt{g}\rangle\texttt{ 2}$$

However, the syntax for the let construct in Figure 4.3 does not allow the variable x to be varied. Therefore, expressions such as e'_D consisting of a choice of variables cannot be described by the syntax given in this section. Similarly, the syntax cannot represent expressions where λ-bound variables are varied as shown in the following expression:

$$e''_{new} \equiv \lambda\ D\langle \texttt{x},\texttt{y}\rangle\texttt{.succ } D\langle \texttt{x},\texttt{y}\rangle$$

Limitations 1 and 2 described above suggest that in order to avoid inconsistency and redefinition of the entire choice calculus functionality, the instantiation of choice calculus for each non-terminal should be avoided. Hence, the syntax and functionality of choice calculus and lambda calculus should be not closely tied. This will ensure that the code related to choice calculus functionality is independent of the modifications to the object language functions and constructs, and vice versa. Limitation 3 indicates that the variational lambda calculus syntax should allow the variable in the let construct and abstractions to be varied. Hence, the variational lambda calculus should be able to

describe the expressions like e'_D and e''_D given above.

The next section discusses the *parameterized choice calculus* approach that overcomes the limitations mentioned above.

4.3 Parameterized Choice Calculus Approach

This section discusses a new approach called *parameterized choice calculus*. A parameterized type takes one or more types as parameters indicated by type variables. Parameterized types are similar to generic or templates types, as these abstract away the details of the type of parameter. For example, in Haskell the `Maybe` type is defined as follows:

```
data Maybe a = Nothing | Just a
```

The parameterized type `Maybe` contains one type variable `a` used in the construct `Just a`. A parameterized type can also contain more than one type parameters as defined in the type `Either` as follows:

```
data Either a b = Left a | Right b
```

The type `Either` takes two type parameters - `a` and `b` that are used in different constructs. Parameterized types with more than one type variables are used when more than one type is required in the constructs.

Parameterized types can also be user-defined, one common example is the definition of the data type `Tree` as follows:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

The type `Tree` is defined as a parameterized type as the tree nodes can store values of any type. The data type `Tree` defines one type variable and it occurs more than once in the `Node` construct indicating that the values to be substituted should be of the same type. Following are some expressions for the above defined datatype:

```
Leaf 1 :: Tree Int

Node 2 (Leaf 1) (Leaf 3) ::  Tree Int

Node "b" (Leaf "a") (Leaf "c") :: Tree [Char]
```

Values of different types cannot be represented by the data type defined above. The following will result in a type error.

```
Node 2 (Leaf 1) (Leaf "b")
```

In this thesis, as the choice calculus can represent variations in any object language, we change the choice calculus to a parameterized type. We change the syntax of choice calculus by replacing *CC* with *CC c* where *c* is a type variable. Parameterized type helps to avoid the re-instantiation of choice calculus for each non-terminal of the object language. Figure 4.5 shows the modified syntax with parameterized data types with *CC e* is replaced with $\hat{e}$ for succinct representation. The modifications in the syntax from the syntax in Figure 4.3 are described as follows:

1. *Separation of object language and variation representation terms*

   This approach separates the lambda calculus and choice calculus constructs. The lambda calculus terms are represented by *e* and the choice calculus as parameterized type is represented by $\hat{c}$(*CC c*). The separation avoids re-instantiation of choice calculus and allows the flexibility to modify lambda calculus without affecting choice calculus and vice versa. We call this

d	::=	$A \mid B \mid \ldots$	*Dimension Name*
t	::=	$a \mid b \mid \ldots$	*Tag Name*
v	::=	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
x	::=	$\texttt{a} \mid \texttt{b} \mid \ldots$	*Variable*
e	::=	$\hat{x}$	*Variable*
	$\mid$	$\lambda \hat{x}.\hat{e}$	*Abstraction*
	$\mid$	$\hat{e}\ \hat{e}$	*Application*
	$\mid$	$\texttt{let}\ \hat{x}{=}\hat{e}\ \texttt{in}\ \hat{e}$	*Let Binding*
	$\mid$	$\hat{e}$	*Parameterized* CC_e
$\hat{c}$	::=	$\textbf{dim}\ d\langle t,\ldots,t\rangle\ \textbf{in}\ \hat{c}$	*Dimension*
	$\mid$	$d\langle \hat{c},\ldots,\hat{c}\rangle$	*Choice*
	$\mid$	$\textbf{let}\ v{=}\hat{c}\ \textbf{in}\ \hat{c}$	*Binding*
	$\mid$	v	*Reference Variable*
	$\mid$	c	*Plain expression*

Figure 4.5: Variation Lambda Calculus-Version 2

independence of changing the object language without having to change the variation representation or vice versa, as *Object Language-Variation Representation independence (OL-VR independence)*.

2. *Parameterized choice calculus*

   The choice calculus defined as parameterized type is represented by $\hat{c}(CC\ c)$ and the similar change is made in all choice calculus constructs. This change avoids re-instantiation of choice calculus allowing any object language non-terminal to be passed as parameter to choice calculus such as *CC e* to add variations to *e*.

3. *Replacement of lambda calculus terms with parameterized CC e terms*

   In order to specify that the variations can be specified in lambda calculus, the plain term *e* is replaced with $\hat{e}(CC\ e)$. The term $\hat{e}$ indicates that a lambda calculus term is passed as parameter to the choice calculus constructs. This allows the specification of variations in the lambda calculus expressions using the choice calculus constructs. Similarly, to allow variations in the variable *x*, it is replaced with $\hat{x}$ in the lambda calculus constructs. Adding variations in this way allows the variations to be specified selectively.

4. *Adding parameterized CC_e to e*

   The above-mentioned changes allow the creation of variational lambda calculus expressions with nested expressions that can be varied, such as $\lambda D\langle \mathtt{x}, \mathtt{y}\rangle.D\langle \mathtt{x}, \mathtt{y}\rangle$. However, these changes cannot create a variational lambda calculus expression that starts with a choice calculus construct, such as $D\langle \lambda\,\mathtt{x}\,.\,\mathtt{x}, \lambda\,\mathtt{z}.\mathtt{z}\rangle$. A new construct *parameterized CC_e* denoted by $\hat{e}$ is added to the term *e* that allows the specification of expressions such as $D\langle \lambda\,\mathtt{x}\,.\,\mathtt{x}, \lambda\,\mathtt{z}.\mathtt{z}\rangle$.

5. *Adding* plain expression *(c) to* $\hat{c}$

   With the change 3 mentioned above, the nested occurrences of *e* in all the lambda calculus

constructs is changed to $\hat{e}$. This implies that the nested expressions in e are choice calculus constructs with e as parameter. As a result, the plain expressions such as $\lambda x.x$ cannot be described with this syntax. Therefore, a new construct *plain expression* denoted by c is added to the choice calculus that allows describing plain expressions in choice calculus.

Advantages:

The advantages of this approach are described as follows:

1. *Object Language-Variation Representation Independence (OL-VR Independence)*

   The above-mentioned approach separates the syntax of an object language from the variation representation i.e. lambda calculus from choice calculus. This independence offers the advantage of modifying the syntax of the object language without the need to modify the syntax of the change representation and vice versa. Additionally, this helps to avoid inconsistencies in the functionality associated with choice calculus.

2. *Selective Variation Introduction*

   With the approach mentioned in this section, variations are introduced in any object language by passing the non-terminal as a parameter to the choice calculus. This provides the flexibility to vary the occurrences of non-terminals selectively. For example, for abstraction in lambda calculus, only the variable can be varied by changing x to $\hat{x}$ keeping the scope of abstraction unchanged.

Limitations:

The limitations of this approach are described as follows:

1. *Non-scalable selective variation introduction*

   Although the above syntax allows the specification of selective variation, it suffers from the problem of scalability. Consider an object language with large number of non-terminals, such

as C. In order to add variations to C, each occurrence of a non-terminal has to be individually modified as a parameter to the choice calculus. This makes the variation introduction process tedious.

Although the parameterized approach provides numerous advantages, scalability is a concerning disadvantage.

4.4 The Scalable Parameterized Choice Calculus Approach

The previous section introduced the parameterized approach with advantages, such as OL-VR independence and selective variation introduction. However, the selective variation introduction imposes a scalability issue for languages with large number of non-terminals.

Reconsidering the syntax introduced in the previous section in Figure 4.5, the choice calculus constructs are introduced in the lambda calculus constructs by the following two changes:

1. Replacing e with $\hat{e}$ on the right-hand side of the production in lambda calculus constructs.
2. Adding a new term *Parameterized CC*$_e$ ($\hat{e}$) in the syntax.

The purpose of the first change was to allow nested expressions to be choice calculus expressions, such as $\lambda D\langle \mathtt{x}, \mathtt{y}\rangle.D\langle \mathtt{x}, \mathtt{y}\rangle$ where variable and expression in an abstraction are choice calculus expressions. And the second change was to allow variational lambda calculus expressions that start with the choice calculus constructs, such as $D\langle \lambda \mathtt{x.succ\ x}, \lambda \mathtt{x.x}\rangle$. Careful examination of the syntax suggests that the expressions with nested choice calculus expressions can also be created only with the second change i.e. parameterized CC_e without the need for first change. This implies that without changing all the occurrences of e to $\hat{e}$, and by only adding the new construct $\hat{e}$, all the possible variations can be represented. Hence, all the $\hat{e}$ nested in the lambda calculus constructs are changed back to e.

A similar change is performed for variable x. Instead of changing x to $\hat{x}$ in e, a new construct *Parameterized* $CC_x(\hat{x})$ is added to the term x.

d	::=	$A \mid B \mid \ldots$	*Dimension Name*
t	::=	$a \mid b \mid \ldots$	*Tag Name*
v	::=	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
x	::=	$\texttt{a} \mid \texttt{b} \mid \ldots$	*Plain Variable*
	$\mid$	$\hat{x}$	*Parameterized* CC_x
e	::=	x	*Variable*
	$\mid$	$\lambda x.e$	*Abstraction*
	$\mid$	$e\ e$	*Application*
	$\mid$	$\texttt{let}\ x\texttt{=}e\ \texttt{in}\ e$	*Let Binding*
	$\mid$	$\hat{e}$	*Parameterized* CC_e
$\hat{c}$	::=	$\textbf{dim}\ d\langle t,\ldots,t\rangle\ \textbf{in}\ \hat{c}$	*Dimension*
	$\mid$	$d\langle \hat{c},\ldots,\hat{c}\rangle$	*Choice*
	$\mid$	$\textbf{let}\ v\texttt{=}\hat{c}\ \textbf{in}\ \hat{c}$	*Binding*
	$\mid$	v	*Reference Variable*
	$\mid$	c	*Plain expression*

Figure 4.6: Variational Lambda Calculus-version 3

Figure 4.6 shows the variational lambda calculus syntax resulting from the changes mentioned above. The variations are added in e by only one construct $\hat{e}$ and in variable x by the construct $\hat{x}$.

Advantages:

The advantages of this approach are described as follows:

1. *Object Language-Variation Representation Independence (OL-VR Independence)*

   Similar to the previous approach this approach also offers the advantage of keeping the object language independent of the variation representation and vice versa.

2. *Scalable Variation Introduction*

The current approach offers the advantage that the variations can be introduced in any object language by adding just one new construct for each non-terminal of the object language. As a result, the overhead of modifying each occurrence of the non-terminal in the syntax is avoided. This is beneficial especially in scenarios where the language has large number of constructs that have to be modified.

Limitations:

The limitations of this approach are described as follows:

1. *Non-Selective Variation Introduction*

   This approach does not provide the flexibility of selectively introducing the variations to each occurrence of the non-terminal in a construct as the variations are now introduced by a single independent construct. However, as seen in the previous section, selectivity of variation introduction at the construct level requires lot of extra effort. Besides, for variational lambda calculus selectivity is not required.

The syntax in Figure 4.6 allows scalable variation introduction to languages with large number of non-terminals while offering OL-VR independence. As the selective variation introduction is a tedious task and is not required by variational lambda calculus, the limitation of this approach is not a concern.

4.5 Restrictive Variation Introduction and Trade-off

Section 4.3 added variations to a variable by two changes. The first change replaced all the occurrences of the variable x with $\hat{x}$, and the second change added a new construct parameterized CC_x to the variable. Section 4.4 illustrated that only with the second change, all the possible variation expressions can be represented by the syntax in Figure 4.6 implying that the first change is

unnecessary.

The construct parameterized CC_x represents passing x as a parameter to choice calculus. As a result, variations in x can range over all the choice calculus constructs.

Consider the following two expressions specified using the syntax in Figure 4.6:

$$e' \equiv \lambda(\texttt{let v=x in v+v}).\texttt{v 2}$$
$$e'' \equiv \lambda(\textbf{dim}\ d\langle t_1, t_2\rangle\ \textbf{in}\ D\langle \texttt{x}, \texttt{y}\rangle).D\langle \texttt{x}, \texttt{y}\rangle$$

Although the expressions *e'* and *e"* can be specified using the syntax, these are not correct. The let and the dimension in place of the variable, make these expressions incorrect, in reference to the variation expressions that we intend to represent. Even though the variations in a variable have to be represented with choice calculus, but only some of the choice constructs are allowed in variable.

Similar cases of restrictive variation introduction can be handled in the following two ways:

1. Change the syntax in a manner that illegal expressions cannot be specified.

2. Keep the syntax unchanged and add a checker that can rule out the illegal expressions.

The first method implies that the syntax in Figure 4.6 is changed by restricting x to be varied only by the allowed choice calculus construct i.e. choice. This is achieved by replacing parameterized CC_x with the construct *variable choice* ($d\langle x, \ldots, x\rangle$) as shown in the syntax in Figure 4.7.

This method offers the advantage that the illegal expressions such as *e'* and *e"* mentioned above cannot be specified with the new syntax. However, this method also implies that the syntax now contains two different choice constructs. First, as choice in the choice calculus syntax and second, in term x as variable choice. Hence, two copies of functionality associated with choice construct have to be maintained. This approach with syntax described in Figure 4.7 violates OL-VR independence and can lead to inconsistencies in the code related to choices constructs.

d	$::=$	$A \mid B \mid \ldots$	*Dimension Name*
t	$::=$	$a \mid b \mid \ldots$	*Tag Name*
v	$::=$	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
x	$::=$	$\texttt{a} \mid \texttt{b} \mid \ldots$	*Variable*
	$\mid$	$d\langle x,\ldots,x\rangle$	*Variable Choice*
e	$::=$	x	*Variable*
	$\mid$	$\lambda x.e$	*Abstraction*
	$\mid$	$e\ e$	*Application*
	$\mid$	$\texttt{let}\ x{=}e\ \texttt{in}\ e$	*Let Binding*
	$\mid$	$\hat{e}$	*parameterized* CC_e
$\hat{c}$	$::=$	$\textbf{dim}\ d\langle t,\ldots,t\rangle\ \textbf{in}\ \hat{c}$	*Dimension*
	$\mid$	$d\langle \hat{c},\ldots,\hat{c}\rangle$	*Choice*
	$\mid$	$\textbf{let}\ v{=}\hat{c}\ \textbf{in}\ \hat{c}$	*Binding*
	$\mid$	v	*Reference Variable*
	$\mid$	c	*Plain Expression*

Figure 4.7: Variational Lambda Calculus-version 4

The second method, keeps the syntax in Figure 4.6 unchanged, so that it does not violate *OL-VR independence*. However, it requires a checker to be written to filter out the illegal expressions.

The choice between either of the two methods imposes a selection between either violating the *OL-VR independence* to create a syntactically correct expressions or delegating the task of checking the correctness of an expression to other algorithms, to have *OL-VR independence*. This thesis chooses the second method of dealing with illegal expressions. Hence, the syntax presented in the Figure 4.6 is selected as the final syntax for variational lambda calculus.

4.6 Haskell Syntax and Examples

```
type Name  =  String
type Dim   =  Name                    Dimension Name
type Tag   =  Name                    Tag Name
type V     =  Name                    Reference Variable
data Var   =  Var Name                Variable
           |  CCVar (CC Var)          Parameterized CC_x
data VLC   =  Use Var                 Variable
           |  Abs Var VLC             Abstraction
           |  App VLC VLC             Application
           |  ELet Var VLC VLC        Let Binding
           |  CCVLC (CC VLC)          Parametrized CC_VLC
data CC c  =  Dim Dim [Tag] (CC c)    Dimension
           |  Chc Dim [(CC c)]        Choice
           |  Share V (CC c) (CC c)   Binding
           |  Ref V                   Reference Variable
           |  Exp c                   Plain Expression
```

Figure 4.8: Variational Lambda Calculus-version 3(Haskell)

The Figure 4.8 presents the Haskell representation of the variational lambda calculus syntax designed using the scalable parameterized choice calculus approach. The data type `Exp` correspond to lambda calculus and the data type *CC c* corresponds to choice calculus. Using these data types,

the expressions e_{old}, e_{new} and e_D defined on page 41 are described as `hold`, `hnew` and `hD` as follows:

```
varx = Var "x"
usex = Use varx
hold = Abs varx (App succ usex)
hnew = Abs varx (App pred usex)
hD   = Dim "D" [old,new]
       (Exp (Abs varx (App (CCExp (Chc "D" [Exp succ,Exp pred])) usex)))
```

Similarly, the expressions e'_{old}, e'_{new} and e'_D are described in Haskell with the help of the following definitions:

```
varf = Var "f"
varg = Var "g"
varx = Var "x"
var2 = Var "2"
usef = Use varf
useg = Use varg
use2 = Use var2

hold' = ELet varf = Abs varx (App succ usex) in (App usef use2) -- e'old
hnew' = ELet varg = Abs varx (App succ usex) in (App useg use2) -- e'new
hD'   = Dim "D" [old,new]                                       -- e'D
        (Exp (ELet (CCVar (Chc "D" [Exp varf,Exp varg]))
                   (Abs varx (App succ usex))
                   (App (CCVlc (Chc "D" [Exp usef, Exp useg])) use2)))
```

4.7 Summary

The previous sections discussed in detail the various approaches to represent the variations in an object language. The design process required combining the constructs of lambda calculus with choice calculus to create variational lambda calculus. For each approach the proposed variational lambda calculus syntax along with its advantages, limitations, and improvements are discussed. Finally, the trade-off discussed in Section 4.5 selects the *scalable parameterized choice calculus approach* for designing variational lambda calculus for this thesis. The syntax selected for variational lambda calculus is given in Figure 4.6 given on page 54.

The following table summarizes the different approaches studied for designing variational lambda calculus along with their advantages and limitations.

Approach	*Advantages*	*Limitations*
Basic	*1.Simple and Easy*	*1.Requires creation of a new language and functionality* *2.Unmanageable and Inconsistent functionality for language with more than one non-terminals* *3.Limited variations*
Parameterized	*1.OL-VR Independence* *2.Selective Variation Introduction*	*1.Non-scalable variation introduction*
Scalable Parameterized CC	*1.OL-VR Independence* *2.Scalable Variation Introduction*	*1.Non-selective variation introduction*
Restrictive Variation Introduction	*1.Specific variation constructs can be applied to a term*	*1.Inconsistent functionality*

Figure 4.9: Summary

Chapter 5 – Inferring Composite Changes

5.1 Introduction

Section 1.4 illustrated that by reporting program differences as composite changes, can potentially be more helpful for the programmers to understand the program differences. As a result, this thesis aims to infer composite changes from the program differences reported as variation expressions. Chapter 4 discusses extensively the representation of the changes in an object language using variation representation.

This chapter discusses the process involved in identifying composite changes from variation expressions. This composite change inference process is divided into the following three steps:

1. The composite changes that have to be inferred are selected. The structure of the variation expression corresponding to the composite changes is explained, along with the atomic changes that are expected. This information is represented as *patterns*.

2. The change inference algorithms are designed using the information in the variation patterns. These algorithms compare a given variation expression is compared with the expected structure for each selected composite change. The similarity between the variation expression and expected structure suggests that atomic changes are result of a composite change.

3. The inferred composite changes are reported by annotating the given variation expression with the change information.

Section 5.2 discusses pattern language that allows the specification for patterns for the variation expressions. Additionally, this section describes variation patterns corresponding to the compos-

ite changes that are inferred in the later sections. Section 5.3 extends the syntax of variational lambda calculus to include annotations for the inferred composite changes in the variation expression. Section 5.4 discusses the design of change inference algorithms that infer composite changes by analyzing the structure and the choices in a variation expression. Additionally, this section also discusses the exception cases for the composite changes and the detection of these cases by change inference algorithms.

5.2 Patterns

Consider the following example, where the expression $e_{renamef.old}$ is changed to $e_{renamef.new}$ by changing `f` to `g`.

$$e_{renamef.old} \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 2}$$

$$e_{renamef.new} \equiv \texttt{let g=}\lambda\texttt{x.succ x in g 2}$$

The variation expression representing the changes between the expressions e_{old} and e_{new} is described as follows:

$$e_{renamef.D} \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \texttt{let}\ D\langle\texttt{f,g}\rangle\texttt{=}\lambda\texttt{x.succ x in}\ D\langle\texttt{f,g}\rangle\ \texttt{2}$$

The expression $e_{renamef.D}$ suggests that function renaming results in a variation expression that is a dimension with two tags and dimension expression containing choices. Additionally, it suggests that the dimension expression is a let expression consisting a choice in the function name and either the same choice exists in the function scope or no occurrence of the old function name exists in the scope.

The above information specifies a relation between function renaming and a variation expression by describing the structure of the expression resulting from the change and the expected atomic changes. This relation can be leveraged for inferring the composite changes from a given variation expression.

Suppose, we need to find whether the following expression corresponds to function renaming:

$$e'_D \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \texttt{let}\ D\langle\texttt{r,s}\rangle\texttt{=}\lambda\texttt{x.}\lambda\texttt{y.x+y in}\ D\langle\texttt{r,s}\rangle\ \texttt{2 3}$$

The dimension expression in e'_D consists of two occurrences of the choice $D\langle\texttt{r,s}\rangle$, one in

the function name and other in the function scope. A comparison of the expression e'_D with the description of variation expression for function renaming given above, results in a match. This match suggests that the expression e'_D represents renaming of the function `r` to `s`.

Consider another example, e''_D that is a dimension consisting of choices with two alternatives. However, the expression e''_D does not match the description of the variation expression given above. The mismatch results from the absence of choice in place of function name in expression e''_D. Hence, the expression e''_D does not correspond to function renaming.

$$e''_D \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \texttt{let r=}\lambda\texttt{x.}\lambda\texttt{y.x+y in}\ D\langle \texttt{r,s}\rangle\ \texttt{2 3}$$

The two examples presented above with expressions e'_D and e''_D illustrate that the description of the variation expression resulting from a composite change can be used to check if a given variation expression corresponds to that particular change.

Hence, for all the composite changes that we intend to infer, we define a relation between the change and the variation expression resulting from it. This relation is a template that specifies the structure of the variation expression describing the expected type of atomic changes and the relation between these atomic changes. These templates that are called *patterns* and a new language called *pattern language* is designed to describe these patterns.

The Subsection 5.2.1 discusses the syntax of the pattern language and discusses some examples of describing patterns in general. In order to differentiate between patterns for plain expressions and variation expression, the latter are called *variation patterns*. With example illustrations Subsection 5.2.2 discusses the design of variation patterns for the composite changes. This section is concluded by the discussion of the variation pattern semantics in the Subsection 5.2.3.

5.2.1 Syntax

d	::=	$A \mid B \mid \ldots$	*Dimension Name*
t	::=	$a \mid b \mid \ldots$	*Tag Name*
v	::=	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
x	::=	$\texttt{a} \mid \texttt{b} \mid \ldots$	*Plain Variable*
	$\mid$	$\hat{x}$	*Parameterized* CC_x
p	::=	x	*Variable*
	$\mid$	$\lambda x.p$	*Abstraction*
	$\mid$	$p\ p$	*Application*
	$\mid$	$\texttt{let}\ x{=}p\ \texttt{in}\ p$	*Let Binding*
	$\mid$	$\hat{p}$	*Parameterized* CC_p
	$\mid$	$C^*[p]$	*One-Many Context*
	$\mid$	$C[p]\ldots[p]$	*One-One Context*
$\hat{c}$	::=	$\textbf{dim}\ d\langle t,\ldots,t\rangle\ \textbf{in}\ \hat{c}$	*Dimension*
	$\mid$	$d\langle \hat{c},\ldots,\hat{c}\rangle$	*Choice*
	$\mid$	$\textbf{let}\ v{=}\hat{c}\ \textbf{in}\ \hat{c}$	*Binding*
	$\mid$	v	*Reference Variable*
	$\mid$	c	*Plain expression*

Figure 5.1: Pattern Language

A pattern is a template that specifies the expected structure of an expression. The purpose of specifying a pattern is to check the similarity of a given expression with the specified pattern. The pattern comparison is used to infer composite changes from a given variation expressions.

To describe these patterns, we design a pattern language shown in Figure 5.1. As the patterns are described for variational lambda calculus expressions, the pattern language syntax is similar to that of variational lambda calculus. The pattern language consists of all constructs of variational lambda calculus, with two additional constructs, namely, *one-many context* and *one-one context*.

The two newly introduced constructs are explained in the examples that follow.

Suppose we want to specify a pattern that matches the following expression *e*:

$$e \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 2}$$

Using the pattern language syntax, the following pattern *p* corresponding to the expression *e* is defined:

$$p \equiv \texttt{let f=}\lambda\texttt{x.succ x in } C^*\texttt{[f]}$$

The pattern *p* specifies a let expression where `f` is defined as λ`x.succ x` and its scope consists of an application of `f` to `2` abbreviated by the use of a context as C^*`[f]`. In a pattern, a *context* represents an expression with holes that have to be substituted with one or more term specified in the context. The expression C^*`[f]` specifies a *one-many context*, with the term `f` to be substituted in the context. A *one-many context* represents an expression with zero of more holes and all the holes have to be substituted with a single term. For example, in the pattern *p* specified above, the context *C* refers to the expression [ ] `2`. By providing the term `f` in the context as C^*`[f]`, the term `f` will be substituted in the expression [ ] `2`, resulting in the expression `f 2`.

A *one-one context* represents an expression with one or more holes where each hole is substituted with a different term. Unlike one-many context, a one-one can have one or more terms and the number of terms specified in the context is same as the number of holes in the expression that context represents. For example, using one-one context the pattern for the above-mentioned expression *e* is described as:

$$p' \equiv \texttt{let f=}\lambda\texttt{x.succ x in } C\texttt{[f][2]}$$

The context *C* in the expression *C*`[f][2]` refers to the expression [ ] [ ]. The substitution of `f` and `2` in the context will result in the expression `f 2`.

Suppose, we want to specify a pattern that restricts the occurrence of a value or expression to one. For example, if we want to specify a pattern that matches the following expression, which starts with `1+` and contains only one occurrence of `let f=λx.succ x in f 2`.

$$\texttt{1+let f=}\lambda\texttt{x.succ x in f 2}$$

We describe the pattern as:

$$p'' \equiv \texttt{1+}C[\texttt{let f=}\lambda\texttt{x.succ x in f 2}]$$

5.2.2 Variation Pattern

A *variation pattern* describes the structure of a variation expression, such as the following pattern p'':

$$p'' \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \texttt{let f=}\lambda\texttt{x.succ x in}\ C^*[D\langle 2,3\rangle]$$

Similar to variation expression, a variation pattern is a dimension representing a change with two tags and a dimension expression consisting of choices. In the pattern p'', the dimension expression consists of the let expression with scope containing a context C and the choice $D\langle 2,3\rangle$ passed to it.

A variation expression, such as e' matches the pattern p'' because it has the same structure and the expression `f` $D\langle 2,3\rangle$ matches the pattern $C^*[D\langle 2,3\rangle]$.

$$e' \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \texttt{let f=}\lambda\texttt{x.succ x in f}\ D\langle 2,3\rangle$$

Any plain expression such as `let f=λx.succ x in f 2` does not match the variation pattern p''.

The patterns described in the examples so far describe the template for the expression along with the values that are expected. For example, the following expressions do not match the pattern *p"*.

dim $D\langle old,new\rangle$ **in** `let g=λx.succ x in g` $D\langle 2,3\rangle$

dim $D\langle old,new\rangle$ **in** `let f=λx.pred x in f` $D\langle 2,3\rangle$

dim $D\langle old,new\rangle$ **in** `let f=λx.succ x in f` $D\langle 3,4\rangle$

INSERT
$Insert(p)$: **dim** $D\langle old,new\rangle$ **in** $D\langle \varepsilon,p\rangle$

DELETE
$Delete(p)$: **dim** $D\langle old,new\rangle$ **in** $D\langle p,\varepsilon\rangle$

UPDATE
$Update(p,p')$: **dim** $D\langle old,new\rangle$ **in** $D\langle p,p'\rangle$

Figure 5.2: Variation patterns for atomic changes.

For the changes that we want to infer, we want to specify patterns that are general. Pattern language syntax allows the specification of such general patterns. For example, a variation pattern specifying an update change is given in Figure 5.2

The variation patterns for the changes are identified using change identifiers. For example, the pattern for update is identified using the change identifier $Update(p,p')$ and the pattern corresponding to update is **dim** $D\langle old,new\rangle$ **in** $D\langle p,p'\rangle$.

The pattern $Update(p,p')$ describes a dimension with dimension name D, two tags *old* and *new*, and the varied expression as choice $D\langle p,p'\rangle$. All the following variation expressions match the

pattern *Update*(p,p') specified above:

dim $D\langle old, new\rangle$ **in** $D\langle 2, 3\rangle$

dim $D\langle old, new\rangle$ **in** $D\langle \lambda x.2 * x, \lambda x.3 * x\rangle$

dim $D\langle old, new\rangle$ **in** $D\langle \lambda x.2 * x,$ `let f=λx.pred x in f 3`$\rangle$

The variation pattern in *Insert*(p) consists of a dimension expression as choice with two options, first, an ε denoting a null expression and second the pattern p. Similarly, *Delete*(p) consists of a choice between a pattern p and an ε, representing that the pattern p has been removed.

The variation patterns corresponding to the composite changes that are inferred in this thesis are listed in Figure 5.3.

Rename Function The relation between the variation expression and the function renaming discussed in the Section 5.2 is formally defined by the variation pattern *RenameF*(f,f') in the Figure 5.3. The pattern specifies a dimension with dimension name D, two tags *old* and *new*, and the dimension expression. The dimension expression is a let expression that consists of a choice $D\langle f,f'\rangle$ in the function name and function scope that is described by the pattern expression $C^*[D\langle f,f'\rangle]$. The expression $C^*[D\langle f,f'\rangle]$ specifies a one-many context with the expression $D\langle f,f'\rangle$ to be substituted in the expression represented by the context C. The *RenameF*(f,f') pattern can now be used for comparison with the variation expressions to check if the expressions represent function renaming. The comparison of the variation expression $e_{renamef.D}$ defined on page 63 with the pattern *RenameF*(f,f') results in a match.

Rename Recursive Function The pattern *RenameRF*(f,f') is similar to the pattern *RenameF*(f,f'), except that it checks for the choice $D\langle f,f'\rangle$ in the function definition as well.

Rename Argument The pattern *RenameA*(x,y) is also similar to the pattern *RenameF*(f,f') with the difference that it specifies renaming of variable for abstraction instead of the let expression.

RENAME FUNCTION
$RenameF(f,f') : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ \texttt{let}\ D\langle f,f'\rangle = p\ \texttt{in}\ C^*[D\langle f,f'\rangle]$

RENAME RECURSIVE FUNCTION
$RenameRF(f,f') : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ \texttt{let}\ D\langle f,f'\rangle = E^*[D\langle f,f'\rangle]\ \texttt{in}\ C^*[D\langle f,f'\rangle]$

RENAME ARGUMENT
$RenameA(x,y) : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ \lambda D\langle x,y\rangle.C^*[D\langle x,y\rangle]$

FOLD FUNCTION
$Fold(p,f) : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ \texttt{let}\ f = p\ \texttt{in}\ C^*[D\langle p,f\rangle]$

UNFOLD FUNCTION
$Unfold(f,p) : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ \texttt{let}\ f = p\ \texttt{in}\ C^*[D\langle f,p\rangle]$

DELETE FUNCTION
$DeleteF(f,p) : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ D\langle \texttt{let}\ f = p\ \texttt{in}\ C^*[f], C^*[p]\rangle$

MOVE FUNCTION UP
$MoveUp(f) : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ D\langle C[\texttt{let}\ f = p\ \texttt{in}\ p'], \texttt{let}\ f = p\ \texttt{in}\ C[p']\rangle$

DISTRIBUTE FUNCTION
$Distribute(f) : \textbf{dim}\ D\langle old,new\rangle\ \textbf{in}\ D\langle \texttt{let}\ f = p\ \texttt{in}\ C[e_1]\ldots[e_n], C[\texttt{let}\ f = p\ \texttt{in}\ e_1]\ldots[\texttt{let}\ f = p\ \texttt{in}\ e_n]\rangle$

Figure 5.3: Variation patterns for composite changes

Fold Function Let us consider the change function fold, where the following expression $e_{fold.old}$ is changed to $e_{fold.new}$. The expression λx.succ x in the scope of the let expression in $e_{fold.old}$ matches the definition of the function f. Hence, λx.succ x is replaced with f in the expression e_{new}.

$$e_{fold.old} \equiv \texttt{let f=}\lambda\texttt{x.succ x in }\lambda\texttt{x.succ x 2}$$

$$e_{fold.new} \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 2}$$

The difference between the above two expressions is described as follows:

$$e_{fold.D} \equiv \textbf{dim}\, D\langle old, new\rangle \textbf{ in } \texttt{let f=}\lambda\texttt{x.succ x in }D\langle\lambda\texttt{x.succ x, f}\rangle\texttt{ 2}$$

A pattern for the above variation expression can be described as follows, where the context C refers to the expression [] 2.

$$p \equiv \textbf{dim}\, D\langle old, new\rangle \textbf{ in } \texttt{let f=}\lambda\texttt{x.succ x in }C^*[D\langle\lambda\texttt{x.succ x, f}\rangle]$$

A generalization of the above-mentioned pattern is given by the pattern $Fold(p,f)$ in Figure 5.3.

Unfold Function As the change unfold is the reverse of the fold operation, the pattern $Unfold(f,p)$ is similar to that of $Fold(p,f)$ but with options in the choice reversed.

Delete Function Next we discuss the change function delete that is similar to the change unfold. However, in function delete the definition of the function is omitted after the function calls are replaced with the function definitions. Suppose, the function f in the following expression is deleted, resulting in the expression $e_{delete.old}$ as follows:

$$e_{delete.old} \equiv \texttt{let f=}\lambda\texttt{x.succ x in f 2}$$

$$e_{delete.new} \equiv \lambda\texttt{x.succ x 2}$$

The variation expression describing the above change is as follows:

$$e_{delete.D} \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ D\langle \texttt{let f=}\lambda\texttt{x.succ x in f 2, (}\lambda\texttt{x.succ x) 2}\rangle$$

The variation pattern for the expression $e_{delete.D}$ can be described as follows:

$$p \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ D\langle \texttt{let f=}\lambda\texttt{x.succ x in }C^*[\texttt{f}], C^*[\lambda\texttt{x.succ x}]\rangle$$

The pattern $DeleteF(f,p)$ in Figure 5.3 shows the generalization of the above pattern.

Move Function Up The next change that is described involves moving the function up in the abstract syntax tree changing the function scope. For example, this change updates the expression $e_{up.new}$ described below to $e_{up.new}$.

$$e_{up.old} \equiv \texttt{1+(let f=}\lambda\texttt{x.x in (f 2)+3)}$$

$$e_{up.new} \equiv \texttt{let f=}\lambda\texttt{x.x in 1+(f 2)+3}$$

In the expression $e_{up.old}$, the let expression that is the second operand of the add operator + is shifted in such a way that the operand `1` and operator + are moved in the scope of the function. A context C that refers to the expression that contains let definition. For the above expression the context C refers to the expression `1+[]`. Here, we use one-one context with just one term indicating that only one hole exists that has to be substituted. The substitution of the expression `let` f=λ`x.x in (f 2)+3` in the context will result in the expression $e_{up.old}$. As a result, for the general case the pattern describing the old expression is described as C[`let` f=p `in` p'] and the new expression is `let` f=p `in` C[p'].

Distribute Function Suppose the function scope contains more than one function applications, as given in the following expression:

$$e_{down.old} \equiv \texttt{(let f=}\lambda\texttt{x.x in (f 2)+(f 3))}$$

Moving the function definition in all the subexpressions results in the following expression:

$$e_{down.new} \equiv \texttt{(let f=}\lambda\texttt{x.x in f 2)+(let f=}\lambda\texttt{x.x in (f 3))}$$

This change is called distribute function, as it moves the function definition to all the subexpressions in the function scope, shown by the variation pattern $Distribute(f)$.

Although this thesis is limited to the composite changes discussed in this section. Using the approach illustrated in the examples mentioned above, the variation patterns for other composite changes can be described.

5.2.3 Pattern Semantics

The semantics of a pattern defines a relationship between a pattern and a set of variation expressions that correspond to the pattern. As described in Subsection 5.2.1 the syntax of pattern language is same as that of variational lambda calculus, with additional two context constructs namely, one-many context and one-one context. Therefore, to check whether a variation expression matches a pattern, the contexts specified in the pattern have to be matched.

This subsection defines the semantics for the variation patterns described in the Subsection 5.2.2. As the patterns specify the structure and not the actual values, a single pattern match to more than one variation expression. The semantics of a pattern is represented as $p \succ e$, where pattern p matches an expression e.

As defined in Subsection 5.2.1, a one-many context represents an expression with one or more holes which are all substituted with the term specified in the context. The semantics of a one-many context is given as follows:

S-ONE-MANY CONTEXT

$$\frac{C^*[p]{=}p' \qquad p' \succ e'}{C^*[p] \succ e'}$$

A context C^*[p] is matched with an expression by first substituting the pattern p in the expression represented by the context C resulting in pattern p'. Next, the resulting pattern p' is matched with an expression e'. This will ensure that all the contexts in p' are matched to an expression so that the expression e' does not contain any context construct. Hence, the expression e' is the variation expression matching the pattern C^*[p].

Similar to the one-many context, the semantics for the one-one context is defined as follows:

S-ONE-ONE CONTEXT

$$\frac{C[p_1]\ldots[p_n]{=}p' \;\; p' \succ e'}{C[p_1]\ldots[p_n] \succ e'}$$

The semantics is same as that of one-many context, except that instead of a single pattern, a list of pattern is first substituted in the context.

With the semantic rules for the two context constructs defined, next we explain how the variation patterns for the composite changes can be mapped to the variation expressions. The semantics for

the function renaming pattern is described as:

S-RENAME FUNCTION

$$\frac{p \succ e \qquad C^*[D\langle f,f'\rangle] \succ e'}{\texttt{let } D\langle f,f'\rangle = p \texttt{ in } C^*[D\langle f,f'\rangle] \succ \texttt{let } D\langle f,f'\rangle = e \texttt{ in } e'}$$

In order to obtain a variation expression that matches the pattern `let` $D\langle f,f'\rangle = p$ `in` $C^*[D\langle f,f'\rangle]$, the pattern p and the context $C^*[D\langle f,f'\rangle]$ are matched to the expressions. The premise of the rule states that the pattern p is matches to an expression e and the context expression $C^*[D\langle f,f'\rangle]$ is matches to the expression e' using the semantic rule for one-many context. The matched values are then substituted in the original pattern, resulting into the expression `let` $D\langle f,f'\rangle = e$ `in` e'. So, a variation expression of the form `let` $D\langle f,f'\rangle = e$ `in` e' for which the premise evaluate to true, match the variation pattern.

Let us consider the pattern `MoveUp(`f`)` defined in Figure 5.3, the semantics for this pattern is described as follows:

S-MOVE FUNCTION UP

$$\frac{C^*[\texttt{let } f = p \texttt{ in } p'] \succ e' \qquad p \succ e \qquad C^*[p'] \succ e''}{D\langle C^*[\texttt{let } f = p \texttt{ in } p'], \texttt{let } f = p \texttt{ in } C^*[p']\rangle \succ D\langle e', \texttt{let } f = e \texttt{ in } e''\rangle}$$

In order find a variation expression that matches this pattern, both the alternatives in the choice are matched to expressions. The contexts $C^*[\texttt{let } f = p \texttt{ in } p']$ and $C^*[p']$ are matched to the expressions e' and e'' respectively. And the pattern p is matched to the expression e. Lastly, the values replace the patterns resulting into expression $D\langle e', \texttt{let } f = e \texttt{ in } e''\rangle$.

Figure 5.4 lists the semantics for all variation patterns defined in the subsection 5.2.2.

S-ONE-MANY CONTEXT

$$\frac{C^*[p]{=}p' \qquad p' \succ e'}{C^*[\mathrm{p}] \succ e'}$$

S-ONE-ONE CONTEXT

$$\frac{C[p_1]\ldots[p_n]{=}p' \qquad p' \succ e'}{C[p_1]\ldots[p_n] \succ e'}$$

S-RENAME FUNCTION

$$\frac{p \succ e \qquad C^*[D\langle f,f'\rangle] \succ e'}{\texttt{let } D\langle f,f'\rangle{=}p \texttt{ in } C^*[D\langle f,f'\rangle] \succ \texttt{let } D\langle f,f'\rangle{=}e \texttt{ in } e'}$$

S-RENAME RECURSIVE FUNCTION

$$\frac{E^*[D\langle f,f'\rangle] \succ e \qquad C^*[D\langle f,f'\rangle] \succ e'}{\texttt{let } D\langle f,f'\rangle{=}E^*[D\langle f,f'\rangle] \texttt{ in } C^*[D\langle f,f'\rangle] \succ \texttt{let } D\langle f,f'\rangle{=}e \texttt{ in } e'}$$

S-RENAME ARGUMENT

$$\frac{C^*[D\langle x,y\rangle] \succ e}{\lambda D\langle x,y\rangle.C^*[D\langle x,y\rangle] \succ \lambda D\langle x,y\rangle.ee'}$$

S-FOLD FUNCTION

$$\frac{p \succ e \qquad C^*[D\langle p,f\rangle] \succ e'}{\texttt{let } f{=}p \texttt{ in } C^*[D\langle p,f\rangle] \succ \texttt{let } f{=}e \texttt{ in } e'}$$

S-UNFOLD FUNCTION

$$\frac{p \succ e \qquad C^*[D\langle f,p\rangle] \succ e'}{\texttt{let } f{=}p \texttt{ in } C^*[D\langle f,p\rangle] \succ \texttt{let } f{=}e \texttt{ in } e'}$$

S-DELETE FUNCTION

$$\frac{p \succ e \qquad C^*[f] \succ e' \qquad C^*[p] \succ e''}{D\langle \texttt{let } f{=}p \texttt{ in } C^*[f], C^*[p]\rangle \succ D\langle \texttt{let } f{=}e \texttt{ in } e', e''\rangle}$$

S-MOVE FUNCTION UP

$$\frac{C[\texttt{let } f{=}p \texttt{ in } p'] \succ e' \qquad p \succ e \qquad C[p'] \succ e''}{D\langle C[\texttt{let } f{=}p \texttt{ in } p'], \texttt{let } f{=}p \texttt{ in } C[p']\rangle \succ D\langle e', \texttt{let } f{=}e \texttt{ in } e''\rangle}$$

S-DISTRIBUTE FUNCTION

$$\frac{p \succ e \qquad C[e_1]\ldots[e_n] \succ e' \qquad C[\texttt{ let } f{=}p \texttt{ in } e_1]\ldots[\texttt{let } f{=}p \texttt{ in } e_n] \succ e''}{D\langle \texttt{let } f{=}p \texttt{ in } C[e_1]\ldots[e_n], C[\texttt{let } f{=}p \texttt{ in } e_1]\ldots[\texttt{let } f{=}p \texttt{ in } e_n]\rangle \succ D\langle \texttt{let } f{=}e \texttt{ in } e', e''\rangle}$$

Figure 5.4: Semantics for change patterns.

5.3 Refactoring Annotations Expressions

The composite change inference process compares a given expression with the variation patterns described for the composite changes. A match suggests that the variation expression represents the composite change that corresponds to the matched variation pattern. To track the inferred changes, the change information is annotated in the variation expression. A variation expression annotated with the change information is called a *refactoring-annotated expression*.

A refactoring-annotated expression is similar to a variation expression, but it includes composite change annotations. Therefore, the current syntax of variational lambda calculus has to be extended to include refactoring annotations. A refactoring annotation should describe complete information about the changes, so that the old and the new variants can be recreated using the annotated information. So, the refactoring annotation should include the name of the inferred change, the changed expression, and the old and new value of the expression that has changed.

The construct for refactoring annotation is defined as follows:

$$ra\langle r, r\rangle \textbf{ in } r.$$

It consists of the name of the composite change (*ra*), the old and the new values as *(r,r)* and the changed expression as (*r*).

Section 5.2 described a variation expression $e_{renameF.D}$ for function renaming that matches the variation pattern *RenameF*(*f*,*f'*) described on page 70. Using the annotation construct defined above, the change information can be annotated as follows:

$$r \equiv \textbf{ dim } D\langle old, new\rangle \textbf{ in } RenameF(\texttt{f},\texttt{g}) \textbf{ in } \texttt{let } D\langle \texttt{f},\texttt{g}\rangle = \lambda \texttt{x.succ x in } D\langle \texttt{f},\texttt{g}\rangle\ \texttt{2}$$

The refactoring-annotated expression *r* is a dimension expression, consisting of a refactoring

annotation with the composite change as *RenameF*, `f` and `g` as the old and the new value corresponding to the change, the changed expression as `let` $D\langle$`f,g`$\rangle$=λ`x.succ x in` $D\langle$`f,g`$\rangle$ `2`.

Although the above expression includes the complete change information, it contains redundant expressions. The old and the new values of the changed expression are part of the refactoring annotation as well as the varied expression. To avoid redundancy, the choices in the dimension expression are replaced by the new value for the composite change. As a result, the expression *r* described above is changed to the following annotated expression:

$$r' \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ RenameF(\texttt{f},\texttt{g})\ \textbf{in}\ \texttt{let g=}\lambda\texttt{x.succ x in g 2}$$

While inferring and annotating the changes, the dimension is not removed, as the expression may contain choices that do not correspond to a composite change.

The syntax of the variational lambda calculus given in Figure 4.6 is extended by adding a parameterized non terminal $\ddot{r}(RA\ r)$ as shown in Figure 5.5. It consists of one construct *refactoring annotation* for describing the inferred change. Additionally, the constructs *parameterized* RA_x represented by $\ddot{x}(RA\ x)$ and *parameterized* RA_e represented by $\ddot{e}(RA\ e)$ are added to x and e, respectively.

d	::=	$A \mid B \mid \ldots$	*Dimension Name*
t	::=	$a \mid b \mid \ldots$	*Tag Name*
v	::=	$\underline{a} \mid \underline{b} \mid \ldots$	*Reference Variable*
ra	::=	$a \mid b \mid \ldots$	*Refactoring Name*
x	::=	$a \mid b \mid \ldots$	*Plain Variable*
	$\mid$	$\hat{x}$	*Parameterized* CC_x
	$\mid$	$\ddot{x}$	*Parameterized* RA_x
e	::=	x	*Variable*
	$\mid$	$\lambda x.e$	*Abstraction*
	$\mid$	$e\ e$	*Application*
	$\mid$	`let` x=e `in` e	*Let Binding*
	$\mid$	$\hat{e}$	*Parameterized* CC_e
	$\mid$	$\ddot{e}$	*Parameterized* RA_e
$\hat{c}$	::=	$\mathbf{dim}\ d\langle t,\ldots,t\rangle\ \mathbf{in}\ \hat{c}$	*Dimension*
	$\mid$	$d\langle \hat{c},\ldots,\hat{c}\rangle$	*Choice*
	$\mid$	$\mathbf{let}\ v{=}\hat{c}\ \mathbf{in}\ \hat{c}$	*Binding*
	$\mid$	v	*Reference Variable*
	$\mid$	c	*Plain expression*
$\ddot{r}$	::=	$ra(r,r)\ \mathbf{in}\ r$	*Refactoring Annotation*

Figure 5.5: Variational Lambda Calculus with Refactoring Annotations

5.4 Change Inference Algorithm

Section 5.2.2 defined variation patterns that describe variation expressions corresponding to the composite changes to be inferred. These variation patterns are used in designing change inference algorithm. A *change inference algorithm* analyzes a given variation expression and based on the structure of the expression performs necessary checks to find if the expression corresponds to a composite change. On finding a match, the algorithm annotates the composite change information in the expression creating a refactoring-annotated expression described in 5.3. Similar to variation patterns, the change inference algorithms are object language dependent. This dependence exists because the change inference algorithms analyze the structure of the expression and checks for the composite changes associated with that structure. Hence, the variation patterns and the change inference algorithm are dependent on the object language syntax.

In addition to inferring composite changes, the change inference algorithm also highlights the exception cases to the inferred refactorings. Consider the following expression that is similar to the variation expression $e_{renamef.D}$ on page 63 that corresponds to function renaming.

$$e'_{renamef.D} \equiv \textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \texttt{let}\ D\langle \texttt{f},\texttt{g}\rangle{=}\lambda\texttt{x.succ x in f 2}$$

The comparison of this expression with the pattern *RenameF*(f,f') results in a match, as the one-many context also matches the expression with no holes. Although the expression corresponds to function renaming, the occurrence of `f` in the scope instead of $D\langle \texttt{f},\texttt{g}\rangle$ is an exception. An *exception* is a contradiction to the composite change performed on the expression. It is necessary to report the exceptions to inform the programmer that the composite change has not been fully performed. Therefore, the change inference algorithm finds such exception cases and annotate these with the required information.

This section explains the design of change inference algorithm in Haskell using the variation patterns described in Subsection 5.2.2. The Subsection 5.4.1 discusses the Haskell representation of the extended variational lambda calculus syntax with refactoring annotations described in Section 5.3. The Subsection 5.4.2 discusses the approach used to designing the change inference algorithm and discusses the top level function that interacts with the different data types that are available. The Subsection 5.4.3 illustrates the inference algorithm for function renaming and the related functions.

5.4.1 Syntax

Chapter 4 selected the syntax shown in Figure 4.6 as variational lambda calculus syntax. Section 5.3 briefly described the change inference process, extended the syntax by including the constructs for the refactoring annotations.

Figure 5.6 shows the Haskell representation of the syntax given in Figure 5.5 on page 79. Most of these data types are discussed in Section 4.7. The new syntax adds a new parameterized data type `RA r` corresponding to the non-terminal $\ddot{r}$ in Figure 5.5. For the constructs $\ddot{x}$ and $\ddot{e}$ in the syntax, the constructors `RAVar (RA Var)` and `RAVLC (RA VLC)` are included in the data type `Var` and `VLC`, respectively.

The previous section presented the expression $e'_{renamef.D}$ consisting an exception to the composite change represented by the expression. As the exceptions are annotated in the similar manner as the inferred changes, the construct refactoring annotation can be used to annotate exceptions as well. The two cases can be differentiated by the name of the change, for example, for inferred change ”RenameF” can be used and for exception case, ”RenameFXcp” can be used. However, in order to have the flexibility to report the refactoring annotations and exception annotations in different ways, we add new constructor *refactoring exception annotation* to the non-terminal $\ddot{r}$, as shown in the Figure 5.7.

```
type Name  =  String
type Dim   =  Name                    Dimension Name
type Tag   =  Name                    Tag Name
type V     =  Name                    Reference Variable
data Var   =  Var Name                Variable
           |  CCVar (CC Var)          Parameterized CC_x
           |  RAVar (RA Var)          Parameterized RA_x
data VLC   =  Use Var                 Variable
           |  Abs Var VLC             Abstraction
           |  App VLC VLC             Application
           |  ELet Var VLC VLC        Let Binding
           |  CCVLC (CC VLC)          Parametrized CC_e
           |  RAVLC (RA VLC)          Parametrized RA_e
data CC c  =  Dim Dim [Tag] (CC c)    Dimension
           |  Chc Dim [(CC c)]        Choice
           |  Share V (CC c) (CC c)   Binding
           |  Ref V                   Reference Variable
           |  Exp c                   Plain Expression
data RA r  =  RA Name (r,r) r         Refactoring Annotation
```

Figure 5.6: Variational Lambda Calculus with Refactoring Annotations(Haskell)

```
type Name  =  String
type Dim   =  Name                      Dimension Name
type Tag   =  Name                      Tag Name
type V     =  Name                      Reference Variable
data Var   =  Var Name                  Variable
           |  CCVar (CC Var)            Parameterized CC_x
           |  RAVar (RA Var)            Parameterized RA_x
data VLC   =  Use Var                   Variable
           |  Abs Var VLC               Abstraction
           |  App VLC VLC               Application
           |  ELet Var VLC VLC          Let Binding
           |  CCVLC (CC VLC)            Parametrized CC_e
           |  RAVLC (RA VLC)            Parametrized RA_e
data CC c  =  Dim Dim [Tag] (CC c)      Dimension
           |  Chc Dim [(CC c)]          Choice
           |  Share V (CC c) (CC c)     Binding
           |  Ref V                     Reference Variable
           |  Exp c                     Plain Expression
data RA r  =  RA Name (r,r) r           Refactoring Annotation
           |  XCP Name (r,r) r          Refactoring Exception Annotation
```

Figure 5.7: Variational Lambda Calculus with Refactoring Annotations(Haskell)

5.4.2 Inference Approach

Before discussing the details of algorithm for individual composite changes, we describe the analysis performed by the top level function of the change inference algorithm.

The function first analyzes the given variation expression and depending on the construct of the expression, it checks for the composite changes that are possible. For example, on encountering a let construct, it checks for composite changes related to let expression. The function performs certain checks to determine if the expression can correspond to the composite change. These checks are based on the information specified in the variation patterns. If the variation expression satisfies the performed check, then the expression is passed to the function corresponding to the detected composite change. These functions checks if the expression matches the variation pattern completely. This mainly involves checking if the variation expression corresponds to the contexts specified in the variation patterns for that change. Then, it annotates the expression with the inferred change. If the given expression does not satisfy the first check that is performed, then the algorithm analyzes the nested expressions and the process is repeated. The analysis of variation expression is done in a top-down manner, annotating the variation expression if necessary while parsing from top to bottom.

The function that analyzes the variation expression first is `inferRefVLC`. This function analyzes the construct of the variation expression by using pattern matching. For each construct, the function further checks if the expression corresponds to a composite change. For example, for an abstraction, it checks if the expression corresponds to the variation pattern for renaming argument. This is done by checking if the variable is a choice consisting of two alternatives. If the initial check is satisfied, then the expression is passed to the function `RenameAVlc` that checks if the rest of the expression corresponds to the variation pattern for renaming argument.

For explaining the change inference process, we present an implementation of the function

`inferRefVlc` that is limited to inferring only function renaming. The argument and the return type of the function is same, suggesting that the expression type remains unchanged. A version of `inferRef` exists for all the data types that exist in the syntax, but the change inference is done only in `inferRefVlc`.

```
inferRefVlc :: VLC -> VLC
inferRefVlc (Use v)               = Use $ inferRefVar v
inferRefVlc (Abs v e)             = Abs (inferRefVar v) (inferRefVlc e)
inferRefVlc (App e e')            = App (inferRefVlc e) (inferRefVlc e')
inferRefVlc (ELet v e e')
    | isRenameFVlc (ELet v e e')  = inferRefVlc $ renameFVlc (ELet v e e')
    | otherwise                   = ELet (inferRefVar v) (inferRefVlc e)
                                         (inferRefVlc e')
inferRefVlc (CCVLC e)             = CCVLC (inferRefCCVlc e)
inferRefVlc (RFVLC e)             = RFVLC (inferRefRFVlc e)
```

Using pattern matching the function determines the construct of the variation expression. As the above given implementation only infers function renaming, for all the constructs other than let, the function simply applies the appropriate `inferRef` function to the nested expressions depending its data type. For example, for the abstraction, the function `inferRefVar` is applied to `v` and function `inferRefVlc` is applied to `e`. As all these functions do not change the data type of the parameters, the nested expression remain of the same type. Similarly, for construct `(CCVLC e)` the `inferRefCCVlc` function is applied to `e` and for `(RFVLC e)` the `inferRefRFVlc`.

If a given expression is a let construct, then the function `inferRefVlc` checks for all the possible refactorings for let construct. One of these refactorings is function renaming, for which an initial check is performed by using the function `isRenameFVlc`. The function `isRenameFVlc` accepts an expression of type `VLC` and returns a boolean value. It checks if the variable in the let expression is a variable choice or a plain variable.

```
isRenameFVlc :: VLC -> Bool
isRenameFVlc (ELet v c c') = isVarChc v
isRenameFVlc e             = False
```

The function `isVarChc` checks whether the variable `v` is a variable choice or not.

If the check evaluates to true, this indicates that the expression corresponds to function renaming and the function `renameFVlc` is applied to the expression. This function matches the scope of the let expression with the variation pattern. If all the checks are satisfied, the choices in the let expression corresponding to function renaming are replaced with the new value of the change, and the resulting expression is annotated with the change information.

The implementation of the `inferRef` function for various data types, is given below:

```
inferRefCCVlc :: CC VLC -> CC VLC
inferRefCCVlc (Exp e)             = Exp $ inferRefVlc e
inferRefCCVlc (Dim d ts e)        = Dim d ts (inferRefCCVlc e)
inferRefCCVlc (Chc d es)          = Chc d (map inferRefCCVlc es)

inferRefCCVar :: CC Var -> CC Var
inferRefCCVar (Exp e)             = Exp $ inferRefVar e
inferRefCCVar (Dim d ts e)        = Dim d ts $ inferRefCCVar e
inferRefCCVar (Chc d es)          = Chc d $ map inferRefCCVar es

inferRefVar :: Var -> Var
inferRefVar (Var v)               = Var v
inferRefVar (CCVar e)             = CCVar $ inferRefCCVar e
inferRefVar (RFVar e)             = RFVar $ inferRefRFVar e

inferRefRFVlc :: RF VLC -> RF VLC
inferRefRFVlc (RA n (v,v') e)     = RA n (v,v') $ inferRefVlc e
inferRefRFVlc (XCP n (v,v') e)    = XCP n (v,v') $ inferRefVlc e

inferRefRFVar :: RF Var -> RF Var
inferRefRFVar (RA n (v,v') e)     = RA n (v,v') $ inferRefVar e
inferRefRFVar (XCP n (v,v') e)    = XCP n (v,v') $ inferRefVar e
```

Although this implementation may suggest that the concepts of generic programming given in [27] can be used here. However, this approach is cannot be used for our implementation and we describe the reason at the end of this section.

Next, we explain the implementation of the change-specific functions by showing the implementation of `renameFVlc`. These functions match a given expression completely with the variation pattern, transform the expression by replacing choices with the new values and annotated the expressions with the information about the inferred change. Additionally, these functions find and

annotate the exceptions, if any to the inferred composite change.

5.4.3 Illustration - Function Renaming

The implementation of the function `renameFVlc` is as follows:

```
renameFVlc :: VLC -> VLC
renameFVlc (ELet v e e')
    | r              = appendRFtoVlc (1,"RenameF",o,n)
                       (ELet (repChcVar ("RenameF",o,n) v)
                             (repChcVlc ("RenameF",o,n) e)
                             (repChcVlc ("RenameF",o,n) e'))
    | otherwise      = ELet v e e'
        where
        (r,o,n) = chkRnmFVlc (ELet v e e')
renameFVlc e         = e
```

For let construct, the function checks if it corresponds to function renaming by using the function `chkRnmFVlc`. This function accepts an expression of type `VLC` and returns a tuple of three values. The first value is a boolean value indicating whether the expression corresponds to function renaming by using the function `isRenameFVlc`. The second value in the tuple is old name of the function and the third value is the new name of the function. The implementation of the function `chkRnmFVlc` is as follows:

```
nvlc = Use (Var "")

chkRnmFVlc :: VLC -> (Bool,VLC,VLC)
chkRnmFVlc (ELet v c c')
    | isRenameFVlc (ELet v c c') = (True,Use (head (getChcAltsVar v))
                                      ,Use (last (getChcAltsVar v)))
    | otherwise                  = (False,nvlc,nvlc)
chkRnmFVlc e                     = (False,nvlc,nvlc)
```

The function `getChcAltsVar` returns the two alternatives of a variable choice in a list. The old and the new value is obtained using the function `head` and `last`. If the function `chkRnmFVlc` returns true, then `renameFVlc` first modifies by let expression by replacing the choices corresponding to the detected function renaming, with the new function name. As the choices exist in

variable and the scope of the let expression, two functions, `repChcVlc` and `repChcVar` are used for modification. The implementation of these two functions, is given below:

```
repChcCCVlc :: (Name,VLC,VLC) -> CC VLC -> CC VLC
repChcCCVlc (ra,o,n) (Exp e)                = Exp $ repChcVlc (ra,o,n) e
repChcCCVlc (ra,o,n) (Chc d cs)
    | (length cs == 2) &&
     (("D",o,n) == (d,oldCCVlc cs,newCCVlc cs)) = last cs
    | otherwise                                 = Chc d cs
repChcCCVlc (ra,o,n) (Dim d ts e)               = Dim d ts $ repChcCCVlc
                                                    (ra,o,n) e
repChcCCVlc (ra,o,n) (Let d e e')               = Let d e  $ repChcCCVlc
                                                    (ra,o,n) e'

repChcCCVar :: (Name,VLC,VLC) -> CC Var -> CC Var
repChcCCVar (ra,o,n) (Exp e)                    = Exp $ repChcVar (ra,o,n) e
repChcCCVar (ra,o,n) (Chc d cs)
    | (length cs == 2) &&
      (("D",o,n) == (d,Use (oldCCVar cs),Use (newCCVar cs))) = last cs
    | otherwise                                 = Chc d cs
repChcCCVar (ra,o,n) (Dim d ts e)               = Dim d ts $ repChcCCVar (ra,o,n) e
repChcCCVar (ra,o,n) (Let d e e')               = Let d e $ repChcCCVar (ra,o,n) e'
```

Both the above functions, check if the choice contains two alternatives and if the value of alternatives is same as given in the tuple. If the values match, instead of returning the choice, the new value is returned.

Additionally, the function `renameFVlc` annotates the modified expression using the function `appendRFtoVlc`. The first argument to the function `appendRFtoVlc` is a tuple of four values, the first integer value is just added to indicate whether to add annotation for refactoring or for exception. The second value specifies the name of the inferred change, and the last two values are the expressions of type `VLC` showing the new and the old value for the inferred change.

```
appendRFtoVlc :: (Int,Name,VLC,VLC) -> VLC -> VLC
appendRFtoVlc (i,ra,Use (Var ""),Use (Var "")) e = e
appendRFtoVlc (i,ra,o,n)                          e
    | i==1                                  = RFVLC (RA ra (o,n) e)
    | i==2                                  = RFVLC (XCP ra (o,n) e)
    | otherwise                             = e
```

The following example illustrates the annotation of a variation expression representing the composite change function renaming. Reconsider the expression $e_{renamef.D}$ given in Section 5.2.

The function `inferRefCCVlc` is applied to the expression $e_{renamef.D}$, which applies the function `inferRefVlc` to the dimension expression. The function `inferRefVlc` performs an initial check on the let expression using the `isRenameFVlc`. As the expression $e_{renamef.D}$ contains a choice in variable, the function `isRenameFVlc` returns true. Next, the function `renameFVlc` finds the old and the new value for function renaming using the `chkRnmFVlc` that returns the tuple `(True,f,g)`. These values are used to replace the choices related to the inferred function renaming in the expression e_D. Function `repChcCCVar` replaces the choice $D\langle \texttt{f},\texttt{g}\rangle$ in variable with `g`, and the function `repChcCCVlc` replaces the same choice in the scope of the let expression with `g`. Finally, the function `appendRFtoVlc` appends the annotation, returning the following expression where the annotated expression is shown with `[]`:

$$\textbf{dim}\ D\langle old, new\rangle\ \textbf{in}\ \mathit{RenameF}(\texttt{f},\texttt{g})\ \textbf{in}\ [\texttt{let g=}\lambda\texttt{x.succ x in g 2}]$$

The above expression shows the difference between two expression in terms of composite change, providing complete information about the change. Hence, the differences when reported as the above expression, instead of reporting as $e_{renamef.D}$ provide a better explanation of the change.

5.4.4 Exceptions

Let us reconsider the expression $e'_{renamef.D}$ given in the beginning of this section. As the dimension expression in $e'_{renamef.D}$ is a let expression, the function `inferRefVlc` checks for function renaming using the function `isRenameFVlc`. The check evaluates to true because of the presence of the choice $D\langle \texttt{f},\texttt{g}\rangle$ in the variable and the function `renameFVlc` replaces the choices with the

new value and annotates the expression.

But, in this expression, the occurrence of `f` in the scope of let is an exception to the composite change that has been done. These exceptions are detected using the function `findXcpinVlc` and annotated using `appendRFtoVlc`. The function `findXcpinVlc` takes as argument a tuple consisting of the name of inferred change, the old and the new values, and looks for the occurrence of the old value in the expression. If it finds the old value, it annotates each occurrence as exception as shown below, with the exception annotated expression in {}.

dim $D\langle old,new\rangle$ **in** *RenameF*(`f,g`) **in** [`let g=`λ`x.succ x in` *RenameFXcp*(`f,g`) **in** {`f`} `2`]

```
findXcpinVlc :: (Name,VLC,VLC) -> VLC -> VLC
findXcpinVlc (ra,o,n) (Use v)
    | isXcpInVar (o,n) v              = appendRFtoVlc (2,ra,o,n) (Use v)
    | otherwise                       = Use v
findXcpinVlc (ra,o,n) (Abs v e)       = Abs (findXcpinVar (ra,o,n) v)
                                            (findXcpinVlc (ra,o,n) e)
findXcpinVlc (ra,o,n) (App e e')      = App (findXcpinVlc (ra,o,n) e)
                                            (findXcpinVlc (ra,o,n) e')
findXcpinVlc (ra,o,n) (ELet v e e')
    | chkRedef (ra,o,n) (ELet v e e') = ELet v e e'   -- checks redefinition
    | otherwise                       = ELet (findXcpinVar (ra,o,n) v)
                                             (findXcpinVlc (ra,o,n) e)
                                             (findXcpinVlc (ra,o,n) e')
findXcpinVlc (ra,o,n) (CCVLC e)       = CCVLC $ findXcpinCCVlc (ra,o,n) e
findXcpinVlc (ra,o,n) (RFVLC e)       = RFVLC $ appXcpRFVlc (ra,o,n) e
```

The exception is checked for variable using the function `isXcpInVar`. If it returns true, then the variable is annotated as an exception using the function `appendRFtoVlc`. Suppose, the renamed function is redefined in the scope, as shown in the following expression:

dim $D\langle old,new\rangle$ **in** `let` $D\langle$`f,g`$\rangle=\lambda$`x.succ x in let f=`λ`x.x in f 2`

The function `findXcpinVlc` checks for redefinition using the function `chkReDef`. On find-

ing a redefinition, the function does not traverse the expression further. As a result, the above expression is annotated as follows:

dim $D\langle old, new\rangle$ **in** *RenameF*(f,g) **in** let g=λx.succ x in f 2

To annotate exceptions in a given expression, the implementation of `renameFVlc` is modified by adding a call to function `findXcpinVlc`, as shown below:

```
renameFVlc :: VLC -> VLC
renameFVlc (ELet v e e')
    | r              = appendRFtoVlc (1,"RenameF",o,n)
                       (findXcpinVlc ("RenameFXcp",o,n)
                            (ELet (repChcVar ("RenameF",o,n) v)
                                  (repChcVlc ("RenameF",o,n) e)
                                  (repChcVlc ("RenameF",o,n) e')))
    | otherwise             = ELet v e e'
       where
       (r,o,n) = chkRnmFVlc (ELet v e e')
renameFVlc e         = e
```

The functions for other composite changes are implemented in a similar manner. For the cases where more than one composite change is possible for a single construct, we check the changes one by one. For example, as most of the composite changes discussed in Section 5.2 are related to functions, the algorithm checks first for function renaming, then fold and so on, as shown below:

```
refVlcR (ELet v e e')
    | isRnmFVlc (ELet v e e') == True         = refVlcR $ rnmfVlc (ELet v e e')
    | isFoldVlc (ELet v e e') == True  ||
      (isFoldVlc (ELet v e e') == False &&
      isFoldXcpVlcTop 1 (ELet v e e')==True)  = foldVlc (ELet v e e')
    | isUFoldVlc (ELet v e e') == True ||
      (isUFoldVlc (ELet v e e') == False &&
      isFoldXcpVlcTop 2 (ELet v e e')==True)  = ufoldVlc (ELet v e e')
      ....
      ....
      ....
```

We tried to simplify the implementation of the change inference algorithm by using the concept of generic programming described in [27]. However, we discovered that the approach was not ap-

propriate for our implementation, because the function `everywhere` uses a bottom-up approach, as opposed to the top-down analysis that is done here. This approach could not process the following expression in the way we expected.

$$\texttt{let } D\langle \texttt{f},\texttt{g}\rangle = \lambda \texttt{x.succ x in let f} = \lambda \texttt{x.x in } D\langle \texttt{f},\texttt{g}\rangle \ \texttt{2}$$

Although the above expression corresponds to renaming, but as the function is redefined in the scope, the choice in the scope of the inner let expression is not part of the function renaming. Performing the redefinition checks while using `everywhere` is not possible.

Chapter 6 – Conclusions and Future Work

6.1 Conclusions

In this thesis we have illustrated the different approaches that can be used to represent the variations in an object language using choice calculus. Next, we have shown the composite change inference from the atomic changes described as variation expressions. We have designed the composite change inference by first describing the variation patterns for the selected composite changes. These variation patterns have then been used to design the composite change inference algorithm that accepts a variation expression as input and annotates it with the composite changes it represents. Additionally, the change inference algorithm also detects the exception cases for each inferred refactoring. The variation expression annotated with the composite change information provides a high-level explanation of the changes described by the variation expression.

6.2 Future Work

Similar to the system designed in this thesis, a change inference process for C language can be designed. Garrido [12] has listed the refactorings pertaining to the language C and has designed the C Refactoring (CR) tool for inferring the listed refactorings.

We have done some initial steps to show that this research direction can be pursued. We have simplified the C syntax given in Language.C.Syntax.AST [1], which is based on the grammar given in [20]. Based on the results from Chapter 4, we have designed a new language called *variational C* using the scalable parameterized choice calculus approach discussed in Section 4.4. Figure 6.1

shows the Haskell syntax of variational C.

Suppose, we want represent the following function using the above syntax:

```
int function f (int x, int y)
{
    return x+y;
}
```

Using the data types defined in the Figure 6.1, this function is given by the variable `fd1` as described below:

```
vx = Var "x"
vy = Var "y"

addxy =  Bin Add vx vy                      -- x + y
parx = Par TInt (DrSL "x")                  -- int x
pary = Par TInt (DrSL "y")                  -- int y
fd1fb = CFB [] [SRet addxy]                 -- function body of fd1
fd1 = FD TInt (DrSL "f") [parx,pary] fd1fb
```

Suppose, argument renaming is performed on the above defined function that changes the parameter name from `x` to `a` and from `y` to `b`. Using the choice calculus, the change should be described as:

```
int function f (int D<x,a>, int D<y,b>)
{
    return D<x,a>+D<y,b>;
}
```

The following Haskell code defines the above-mentioned change as `fd2`:

```
vx = Var "x"
vy = Var "y"
va = Var "a"
vb = Var "b"
-- Variation Expressions
chcxa = Chc "D" [Exp vx,Exp va]                          -- d<x,a>
chcyb = Chc "D" [Exp vy,Exp vb]                          -- d<y,b>
addxayb = Bin Add (CCE chcxa) (CCE chcyb)                -- d<x,a> + d<y,b>
parxa = Par TInt (CCDr (Chc "D" [Exp (DrSL "x"), Exp (DrSL "a")]))-- int d<x,a>
paryb = Par TInt (CCDr (Chc "D" [Exp (DrSL "y"), Exp (DrSL "b")]))-- int d<y,b>
fd2fb = CFB [] [SEx addxayb]                             -- function body(fd2)
fd2 = FD TInt (DrSL "f") [parxa,paryb] fd2fb
```

```
data CCon = I Int | CH Char             -- Constants
data CSL = SL String                    -- String Lit
data CUOp = AOp                         -- Unary Op Assign
data CBOp = Add                         -- Binary Op Add
  | Sub                                 -- Binary Op Subtract
  | CCBO (CC CBOp)                      -- Parameterized CC CBOp
data CType = TVoid                      -- Void
  | TChar                               -- Char
  | TInt                                -- Int
  | CCT (CC CType)                      -- Parameterized CC CType
  | RFT (RF CType)                      -- Parameterized RF CType
data CEx = Ass CUOp CEx CEx             -- Assign Expressions
  | Bin CBOp CEx CEx                    -- Binary Expressions
  | Var String                          -- Variable
  | Const CCon                          -- Constant
  | Stmt CStmt                          -- Statement
  | CCE (CC CEx)                        -- Parameterized CC CEx
  | RFE (RF CEx)                        -- Parameterized RF CEx
data CStmt = SEx CEx                    -- Expressions
  | SIf CEx CStmt                       -- If statement
  | SRet CEx                            -- Return statement
  | SRetE                               -- Empty Return
  | CCS (CC CStmt)                      -- Parameterized CC CStmt
  | RFS (RF CStmt)                      -- Parameterized RF CStmt
data CDeclr = DrSL String               -- Variable Name
  | DrA String Int                      -- Array Name and size
  | CCDr (CC CDeclr)                    -- Parameterized CC CDeclr
  | RFDr (RF CDeclr)                    -- Parameterized RF CDeclr
data CDecl =  DlDr CType CDeclr         -- Variable Declaration
  | DlEx CType CEx                      -- Variable Declaration with expression
  | CCDl (CC CDecl)                     -- Parameterized CC CDecl
  | RFDl (RF CDecl)                     -- Parameterized CC CDecl
data CPar = Par CType CDeclr            -- Paramater Declaration
  | CCP (CC CPar)                       -- Parameterized CC CPar
  | RFP (RF CPar)                       -- Parameterized RF CPar
data CFun = FD CType CDeclr [CPar] CFB  -- Function Definition
  | CCFD (CC CFun)                      -- Parameterized CC CFun
  | RFFD (RF CFun)                      -- Parameterized RF CFun
data CFB = CFB [CDecl] [CStmt]          -- Function Body
data CExtDecl = EDF CFun | EDD CDecl    -- External declarations
data CTU = TU [CExtDecl]                -- Translation Unit
data CA = CFun CFun                     -- C Annotation data types
  | CPar CPar
  | CEx CEx
  | CStmt CStmt
  | CDecl CDecl
  | CType CType
  | CS String
data RF e = RA Name (CA,CA) e           -- Refactoring Annotation
  | XCP Name (CA,CA) e                  -- Exception Annotation
```

Figure 6.1: Variational C (Haskell)

The change inference algorithm for argument renaming first checks for choice in the parameter list passed to the function definition. If no choice exists in the parameter list, this indicates that the arguments have not been renamed. Presence of choice indicates argument renaming, which results in annotation of the change to the function, along with replacement of each corresponding choice with the new value of parameter.

The following function `CFun` checks for the construct `FD` using pattern matching. Using the function `getStrPair`, it finds a list of pair of new and old values for parameter renaming. Using the functions `repPar` and `repCFB` for parameter and function body, the choices are replaced with the new value of the parameter name. The function `appRALst2CFun` appends the function with the list of parameter renamings.

```
inferRnmPar :: CFun -> CFun
inferRnmPar (FD t nm ps b)
 | ls /=[]   =  appRALst2CFun (reverse $ getRnmRAList ls)
                              (FD t nm (map (repPar ls) ps) (repCFB ls b))
 | otherwise = FD t nm ps b
   where
    ls = getStrPair (map chkRnmPar ps)
```

The function definition of the functions used above is given below:

```
chkRnmPar :: CPar -> (Bool,String,String)
chkRnmPar (CCP _)                    = (False, "", "")
chkRnmPar (Par ts (CCDr (Chc d cs))) = (True,
                                        getDrStr $ head (map remExp cs),
                                        getDrStr $ last (map remExp cs))
chkRnmPar (Par ts _)                 = (False, "", "")

getRnmRAList :: [(String, String)] -> [(Name, Dim, CA, CA)]
getRnmRAList []          = []
getRnmRAList ((s,s'):sl) = ("RenameA", "D", CS s, CS s'):getRnmRAList sl

isVarChc :: CC CEx -> Bool
isVarChc (Chc d (Exp (Var v):[cs])) = True
isVarChc (_)                        = False

getStrPair :: [(Bool,String,String)] -> [(String,String)]
getStrPair ([])  = []
getStrPair ((t,o,n):ps)
```

```
 | t             = (o,n):getStrPair ps
 | otherwise     = getStrPair ps
```

The following are the functions that replace the choices with the new value of the change.

```
repPar :: [(String, String)] -> CPar -> CPar
repPar ls (Par ts (CCDr (Chc d cs)))
 | any (==(getDrStr $ head (map remExp cs), getDrStr $ last (map remExp cs))) ls
            = Par ts (DrSL (getDrStr $ last (map remExp cs)))
 | otherwise = Par ts (CCDr (Chc d cs))

repCFB :: [(String, String)] -> CFB -> CFB
repCFB ls (CFB ds ss) = CFB (map (repCDL ls) ds) (map (repCStmt ls) ss)

repCDL :: [(String, String)] -> CDecl -> CDecl
repCDL ls (DlEx ts e) = DlEx ts (repCEx ls e)
repCDL ls e = e

repCStmt :: [(String, String)] -> CStmt ->  CStmt
repCStmt ls (SEx e)   = SEx $ repCEx ls e
repCStmt ls (SIf e s) = SIf e (repCStmt ls s)
repCStmt ls (SRet e)  = SRet $ repCEx ls e
repCStmt ls  e        = e

repCEx :: [(String, String)] -> CEx -> CEx
repCEx ls (Ass o e e') = Ass o (repCEx ls e) (repCEx ls e')
repCEx ls (Bin o e e') = Bin o (repCEx ls e) (repCEx ls e')
repCEx ls (Stmt s)     = Stmt $ repCStmt ls s
repCEx ls (CCE (Chc d cs))
 | isVarChc (Chc d cs) &&
   any(==(getVarStr $ head (map remExp cs), getVarStr $ last (map remExp cs))) ls
                      = last (map remExp cs)
 | otherwise          = CCE (Chc d cs)
repCEx ls e           = e
```

The functions that append the annotations are implemented as follows:

```
appRALst2CFun :: [(Name,Dim,CA,CA)] -> CFun -> CFun
appRALst2CFun [] f = f
appRALst2CFun ls f = foldl (flip appRA2CFun) f ls

appRA2CFun :: (Name,Dim,CA,CA) -> CFun -> CFun
appRA2CFun ("",d,e,e') f = CFun
appRA2CFun (nm,d,e,e') f = RFFD (RA nm (e,e') f)
```

The above implementation illustrates that the composite change inference from the representation of atomic changes can be done in variational C.

With the change inference system implemented for lambda calculus and C, this system can be evaluated against the existing code for lambda calculus and C. Additionally, an experimental study could be done with the users to evaluate the improvement of a programmer's understanding of the program changes with the system described in this thesis. This study can also determine if the annotations required more information, such as programmer's checked-in messages.

Another area that we want to extend this work is implementing a user interface for the system described in this thesis. The interface should allow the representation of the changes, along with the ability to select the alternatives from the choices. Additionally, it should be able to infer the composite changes from the variation expressions and annotate these changes using color coding or tool tips.

Le [28] has surveyed several activities related to software refactorings. One of these activities is composition of refactorings proposed by Kniesel et al. [25]. They have proposed the idea of a refactoring editor that allows the user to create, edit and compose new refactorings from existing ones. As this thesis provides a structured way of representing and inferring changes, this work could be extended in the area of composition of refactorings.

Bibliography

[1] C syntax. `http://hackage.haskell.org/packages/archive/language-c/0.3.1.1/doc/html/Language-C-Syntax-AST.html`.

[2] Don Batory. Feature models, grammars, and propositional formulas. In *Feature models, grammars, and propositional formulas*, pages 7–20. Springer, 2005.

[3] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[4] Andrei Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES97*, pages 21–29. IEEE Computer Society, 1997.

[5] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, SIGMOD '96, pages 493–504, New York, NY, USA, 1996. ACM.

[6] S. Chen, M. Erwig, and E. Walkingshaw. A Type System for Variational Lambda Calculus. 2011. Submitted for publication.

[7] Serge Demeyer, Stphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *IN PROCEEDINGS OF OOPSLA 2000 (INTERNATIONAL CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS*, pages 166–177. ACM Press, 1999.

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *In D. Thomas, editor, ECOOP, volume 4067 of Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[9] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21:6:1–6:27, December 2011.

[10] Martin Fowler. Catalog of refactorings. http://martinfowler.com/refactoring/catalog/index.html.

[11] Martin Fowler. *Refactoring: Improving the Design of the Existing Code*. Addison-Wesley, Reading, Massachusetts, 1999.

[12] A. Garrido. Software refactoring applied to c programming language. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[13] GNU. The c preprocessor. http://gcc.gnu.org/onlinedocs/cpp. Free Software Foundation.

[14] M. W. Godfreym and L. Zou. Using origin analysis to detect merging and splitting of source code entities. In *IEEE Trans. on Software Engineering, vol. 31*, pages 166–181. IEEE Trans. on Software Engineering, 2005.

[15] Carsten Görg and Peter Weisgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 205–214, Washington, DC, USA, 2005. IEEE Computer Society.

[16] Abram Hindle, Daniel M. German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 99–108, New York, NY, USA, 2008. ACM.

[17] D.S. Hirschberg. *The Longest Common Subsequence Problem*. PhD thesis, Princeton, 1975.

[18] Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 234–245, New York, NY, USA, 1990. ACM.

[19] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM.

[20] Brain W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Reading, Massachusetts, 1988.

[21] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 371–372, New York, NY, USA, 2010. ACM.

[22] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, Washington, DC, USA, 2009. IEEE Computer Society.

[23] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Günter Kniesel and Helge Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52:9–51, August 2004.

[26] AT & T Bell Laboratories, J.W. Hunt, and M.D. McIlroy. *An algorithm for differential file comparison*. Computing science technical report. Bell Laboratories, 1976.

[27] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

[28] Duc Le. Software refactoring: A survey of activities, techniques, and formalisms. Oregon State University, 2011. Ph.D. Qualifier Paper.

[29] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[30] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.

[31] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, MSR '05, pages 1–5, New York, NY, USA, 2005. ACM.

[32] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[33] D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2:1–9, January 1976.

[34] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[35] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction. In *Proceedings of the 2008 international working conference on Mining software repositories*, MSR '08, pages 35–38, New York, NY, USA, 2008. ACM.

[36] Romain Robbes. Mining a change-based software repository. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 15–, Washington, DC, USA, 2007. IEEE Computer Society.

[37] K. D. Volder. *Type oriented Logic Meta Programming*. PhD thesis, University of British Columbia, 1998.

[38] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[39] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21:739–755, 1991.

[41] Wuu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical report, 1989.

[42] Thomas Zimmermann and Peter Weisgerber. Preprocessing cvs data for fine-grained analysis. In *In Proc. International Workshop on Mining Software Repositories*, pages 2–6, Edinburgh, Scotland, U.K., 2004.