AN ABSTRACT OF THE PROJECT DESCRIPTION OF

Keying Xu for the degree of Master of Science in Computer Science presented on August 3, 2015.

Title: Representing Program Edits with the Choice Calculus

Abstract approved: ______________________________

Martin Erwig

The problem of supporting more advanced selective undo operations has received a lot of attention. However, selective undo is generally missing in commonly used editors. Moreover, partial selective undo, the ability of undoing just part of some edit so that other edits may be undone, is not supported at all. We observe that a fundamental obstacle is the lack of a more flexible and compositional edit model. This project addresses this issue and proposes the *choice edit model*, which is based on the representation provided by the choice calculus. The central idea is to represent an edit through a choice that contains the old and the new code as alternatives. Edits inherit properties from choices and can thus be composed, nested, and transformed so that dependent edits may be untangled and undone partially. The choice representation is an internal representation, not meant to be exposed to programmers directly. To communicate the structure and dependencies of edits we introduce program edit graphs as an alternative, more abstract representation.

Program edit graphs explicitly represent program variants and their relations. We also discuss the scalability of PEGs.

Representing Program Edits with the Choice Calculus

by

Keying Xu

A PROJECT DESCRIPTION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented August 3, 2015
Commencement June 2015

ACKNOWLEDGEMENTS

First of all, I’d like to express my thanks to my husband, Sheng, for his great help and encouragement. It’s my greatest pleasure to have been with him for the past thirteen years.
I’m also heartfelt grateful to my advisor, Professor Martin Erwig. His patient advice, conscientious attitude in doing research, and his professional dedication influence me deeply.
Many thanks to my parents and my mother-in-law to help me take care of my babies and support me a lot. Without their help, definitely I would not be able to conduct my study.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (Continued)

Chapter 1: Introduction

In this project description, I present my work on a choice-based edit model that supports flexible program edit, undo, and redo operations. In particular, it can untangle dependent edits to support partial selective undo, an operation that hasn't been supported by previous editors despite numerous efforts of improving them.

This chapter motivates the needs for a better support of program editing, discusses previous work, and outlines the structure of the rest of this project report.

1.1 Flexible undo operations and challenges

Editing program source code and related artifacts is an integral task of software development. However, editing support is still primitive, even with modern editors. Most editors build upon on the linear edit model, which orders all editing operations based on the time points they occurred. While this model is amenable to a simple implementation, it limits editing operations that users may perform. Specifically, this ordering limits the flexibility of undo and redo operations. Earlier operations can be undone only after later operations have been undone. A similar restriction applies for redo operations.

However, it's quite common that we want to keep later edits but undo previous edits. This kind of operations is known as selective undo. A concrete example and a detailed description of selective undo are given in Section 2.2. According to a recent study performed by Yoon and Myers [2014], about 9.5% of all programmer backtracking is selective undo. Selective undo is an active software engineering research topic [Cass and Fernandes, 2007; Myers and Kosbie, 1996; Shao et al., 2010; Yoon and Myers, 2012; Yoon et al., 2013].

In search for smart merging strategies, many version control systems (VCSs) maintain non-linear editing histories, which enable them to support selective undo to some degree. Examples are Git[1], Mercurial[2], and Darcs[3]. However, it's hard to rely on VCSs to provide selective undo for two reasons. First, it's very cumbersome to use VCS to track each editing operation. Second, selective undo fails in many cases in VCSs. To illustrate, consider a very simple example. Assume we have a file that contains two lines: `a` in the first line and `b` in the second. Now we have two changes and correspondingly two commits. The first change modifies `a` to `c` and is named commit A. The second change modifies `b` to `d` and is named commit B. Since A and B change different places of the file, we would expect to be able to undo A without affecting B. However, undoing A will lead to a conflict in both Git and Mercurial, and manual interventions are needed to resolve the conflict. In this situation, Darcs is able to undo the edit A without affecting B.

While supporting selective undo can be challenging, daily editing calls for even more flexible operations, such as the *partial selective undo*, which is explained in Section 2.3. Supporting selective undo and partial selective undo requires an edit model that possesses two properties. First, it must represent program edits in a succinct and compositional way so that different edits can be easily put together and certain edits can be singled

[1]http://git-scm.com
[2]http://mercurial.selenic.com
[3]http://darcs.net/

out for undoing or redoing without affecting other edits. Second, it must be possible to discover relations between edits based on the representation so that entangled edits may be unchained and undone. Besides these properties, the relations between edits and the result of applying edits should be exposed to the user in a simple way so that the benefits of the flexible model can be reaped by the user.

This project report proposes a choice edit model that exhibits both properties and program edit graphs that show the relations between applying program edits and the resulting programs in a systematic way. The choice edit model employs *choices* to represent program edits. Specifically, if a program part p is changed to q, then a choice between p and q is created. Each choice has two alternatives, denoting the "old" program and "new" program, respectively. When multiple places are changed, multiple choices are created. The choice calculus [Erwig and Walkingshaw, 2011] provides a set of rules for transforming choice expressions, which can be used directly to manipulate program edits. Though simple, choice expressions maintain enough structural information so that the relations among edits can be extracted and reasoned about.

1.2 Project report outline

This section gives an overview of the rest of this project report and presents main contributions. This project report is mostly based on the draft paper [Erwig et al., 2015] and extends it with a full description of an empirical study about grouping low-level edits to high-level edits.

Chapter 2 (Background) presents the basis for understanding this report. It gives a detailed account of the linear edit model, selective undo, and partial selective undo through several examples. It also introduces the choice-based variation representation.

Chapter 3 (Choice Edit Model) discusses how choices are introduced as programs are changed and how choice expressions are interpreted as program edits. A very important observation there is that editing programs corresponds to creating program families, which are programs encoding a set of related programs. This chapter also shows two important kinds of relations among program edits, contingent edits and divergent edits, and a set of rules for discovering them.

Chapter 4 (Program Edit Graphs) presents a graphical representation that shows the relations between program families, applying edits, and the resulting programs without exposing choice expressions, the underlying representations of the edit model, to the user. This chapter also presents an algorithm for constructing such graphs and discusses the implications of the density of graphs. This chapter also briefly presents a method to deal with the scalability problem of program edit graphs.

Chapter 5 (Scalability) investigates the scalability problem empirically. Through an empirical study of the Python source code, this chapter concludes that edits in over 85% of files can be grouped into fewer than 4 high-level edits. This demonstrates that program edit graphs can work well in many cases.

Chapter 6 (Conclusion) summarizes this project report and presents several directions for future work.

1.3 Previous work

The desire for formalizing undo operations and supporting more flexible undo operations has a long history. [Leeman, 1986] gave a formalization of linear undo, which means that earlier operations can't be undone without undoing later ones. The insight of that work was that undo can be added to not only program edits, but also any form of computations through a generic form and using a set of only four primitives. A common definition of selective undo was given in [Berlage, 1994], which also presented an approach to selective undo for graphical user interfaces.

Although not termed selective undo, some form of selective undo was introduced by [Archer et al., 1984], who also included an implementation named COPE. However, the model doesn't deal with editing directly but was about transformation of states of other objects, for example, the cursors in editors. The model consists of two kinds of commands: normal commands for manipulating the transformation of objects and metacommands for controlling the executions of normal command sequences, which were called scripts. The selective undo facility was supported through the use of metacommands **undo**, **modify** and **redo**.

undo back to c_i then **modify** commands $\ldots$ in R and then **redo** through c_j.

where c_i is a command and R is a sequence of commands. The form of supported selective undo in COPE was limited since it didn't track dependencies between commands and requires direct interaction between the user and the script being executed. This system was improved in [Vitter, 1984] by adding a mechanism for handling skip commands and in [Yang, 1988] by adding a mechanism for rotating a sequence of commands. Each addition made the original system more powerful, but also incurred more burdens for users since they have to step through commands or explicitly control the executions of commands.

Besides the script-based undo model [Archer et al., 1984] and its derivatives [Vitter, 1984; Yang, 1988], many other models for supporting selective undo have been developed [Zhou and Imamiya, 1997; Myers and Kosbie, 1996; Edwards et al., 2000; Cass and Fernandes, 2007]. The nonlinear undo model [Zhou and Imamiya, 1997] distinguished itself by introducing the notion of local objects so that selective undo can be applied to specific objects while previous models only had global objects. A particular challenge was partitioning global objects, which may contain circular dependencies, into subdirectories to hold local objects. The focus of [Cass and Fernandes, 2007] was to correctly handle dependencies among different user actions and ensure the correct ordering of undoing dependent edits. This model was called *cascading* selective undo. The model distinguishes between parent actions and child tasks and classifies the relations of child tasks to different categories to capture their dependencies.

A large body of work on selective undo for collaborative systems has been done based on the idea of operational transformation [Prakash and Knister, 1994; Ressel and Gunzenhäuser, 1999; Sun, 2002; Weiss et al., 2009; Sun and Sun, 2009; Shao et al., 2010]. Selective undo is critical for collaborative systems, where multiple people may work on the same document and it's unrealistic to undo later edits that may be performed by other people to undo an earlier edit. A particular challenge in collaborative editing is that locations relevant to local editors become invalid when edits are put together and applied to the global document. A common solution to this problem is operational transformation, which adjusts the local locations when edits are combined. To undo the given operation o within the edit history H, [Prakash and Knister, 1994] first transposes o to the end of H by commutating adjacent operations. Note that o will be changed during the transposing process, in particular, the location information needs to be adjusted. When o is transposed to the end of H, it can be undone as usual. The transposing process fails if two operations fail to transpose due to conflicts. [Ressel and Gunzenhäuser, 1999] performs undo in a similar manner but is restricted to local operations only. The improvement is that it achieves convergence of undo operations under certain conditions. Other work in this context mainly focused on showing the properties of undo operations and improving the complexity of undo.

Supporting selective undo in text editors brings up many tricky user interface issues [Yoon and Myers]. The first question results from regional conflicts among edit operations. This can happen when multiple operations are involved in overlapping locations. Another issue is how users specify what to selectively undo since it is difficult to name editing operations and showing snapshots is not helpful either. Azurite [Yoon and Myers, 2011, 2012; Yoon et al., 2013; Yoon and Myers], an Eclipse plugin, addresses these problems by remembering conflicting regions, visualizing the fine-grained editing history through a timeline, allowing users to move back and forward on the timeline, and accepting changes at any time point. Azurite, however, is still based on the linear history model. The main difference between Azurite and the work here is that Azurite focuses on providing a better user interface and is less concerned with the underlying edit model, which is the focus of this work. Thus our work can complement their work.

In this work we are also interested in knowing how many low-level (primitive) edits may be classified into one high-level (conceptual) edit and how many conceptual edits typically appear between two versions of a file. A more detailed definition of primitive and conceptual edits is given in Section 5.1. Many efforts have been made to identify high-level changes from low-level edits [Demeyer et al., 2000; Ying et al., 2004; Dagit and Sottile, 2013; Negara et al., 2014; Kim and Notkin, 2009; Nguyen et al., 2010; Kim et al., 2013]. In many cases, low-level changes are intended to form refactorings, fix bugs, add new features, and so on. Most of these approaches [Demeyer et al., 2000; Ying et al., 2004; Negara et al., 2012b] identify refactorings from low-level edits, such as variable renaming, value extractions, and method renaming. LSdiff [Kim and Notkin, 2009; Nguyen et al., 2010; Kim et al., 2013] works differently in that no predefined rules of grouping changes are specified. Instead, high-level changes together with rules for grouping them are deduced from low-level changes. Most of these approaches are based on mining change histories. Although the completeness and precision of this method are challenged in [Negara et al., 2012b], it remains mainstream for understanding software changes.

Chapter 2: Background

This chapter presents examples and principles that will be used throughout this report. In Section 2.1, I introduce three editing scenarios. In Section 2.2, I present the linear edit model, which doesn't support selective undo and redo. In Section 2.3, I describe entangled edits and partial selective undo. In Section 2.4, I introduce the choice calculus [Erwig and Walkingshaw, 2011], which is a language for representing variations and which forms the basis for the choice edit model.

2.1 Three editing scenarios

I will introduce three editing scenarios that will be used frequently in this project report. The first scenario involves independent edits. Figure 2.1 shows this editing process and the related programs. This scenario starts with the function P_{ab}. Now we rename the parameter `a` to `c`, which yields P_{cb}. We call the change from P_{ab} to P_{cb} edit A. Next we change the local variable `b` to `d`. We obtain the program P_{cd} and refer to this change as edit D.

The second scenario, as shown in Figure 2.2, is a slightly modified version of the first scenario. After the edit A, we change the local variable `b` to `c`, which causes a name conflict between the function parameter and the local variable. To avoid this problem

$$\begin{tikzcd}
\texttt{int f(int a)\{int b; return a+b;\}} \arrow[d, "\text{edit } A"] & \mid\ P_{ab} \\
\texttt{int f(int c)\{int b; return c+b;\}} \arrow[d, "\text{edit } D"] & \mid\ P_{cb} \\
\texttt{int f(int c)\{int d; return c+d;\}} & \mid\ P_{cd}
\end{tikzcd}$$

Figure 2.1: Scenario 1: independent changes.

$$\begin{tikzcd}
\texttt{int f(int a)\{int b; return a+b;\}} \arrow[d, "\text{edit } A"] & \mid\ P_{ab} \\
\texttt{int f(int c)\{int b; return c+b;\}} \arrow[d, "\text{edit } B"] & \mid\ P_{cb} \\
\texttt{int f(int d)\{int c; return d+c;\}} & \mid\ P_{dc}
\end{tikzcd}$$

Figure 2.2: Scenario 2: entangled changes.

we have to rename the parameter `c` to `d`. We refer to this change as edit B and call the resulting program P_{dc}.

The third scenario including three edits is also based on the first scenario. After the edit A, we revert it and modify `a` to `x` at the same time. We view the whole action as edit B' and name the resulting variant P_{xb}. In the following edit C, we delete the local variable `b` and use the constant `5` to substitute the now dangling reference `b`. With that, we obtain the program P_{x5}. We show this editing process in Figure 2.3.

2.2 The linear edit model and selective undo and redo

The linear edit model (LEM) is usually implemented through a use of stacks. Since stack operations involve the top elements only, the LEM allows us to undo the most recent edit

$$\begin{tikzcd}
\texttt{int f(int a)\{int b; return a+b;\}} \arrow[d, "\text{edit } A"] & \mid \; P_{ab} \\
\texttt{int f(int c)\{int b; return c+b;\}} \arrow[d, "\text{edit } B'"] & \mid \; P_{cb} \\
\texttt{int f(int x)\{int b; return x+b;\}} \arrow[d, "\text{edit } C"] & \mid \; P_{xb} \\
\texttt{int f(int x)\{ return x+5;\}} & \mid \; P_{x5}
\end{tikzcd}$$

Figure 2.3: Scenario 3: three edits.

only. Figure 2.4 illustrates the result of undoing edits in scenario 1. We observe that we can successfully undo *D*, the most recent edit made. However, undoing *A* after edit *D* is blocked because *A* is not the most recent edit.

Arguably, it's too restrictive to block the attempt of undoing *A* solely because the edit *D* hasn't been undone. We observe the edit *A* of renaming the parameter is indeed independent of the edit *D* of renaming the local variable. If two edits do not overlap, they are independent of each other, and either one can be undone. We call such kind of undo *selective undo*. The idea of selective undo lifts the restriction of temporal ordering of edits. Figure 2.5 depicts undoing edits in selective undo for the scenario 1. We observe that undoing either edit first is allowed. Moreover, different undoing orderings lead to the same resulting program.

Next we will discuss the selective redo. Based on scenario 3, now assume we want to return to the state of applying the *A* edit but keeping the *C* edit. What we can do is undo the *B′* edit and redo the *A* edit. Although edit *C* occurs after both the edits *A* and *B′*, we

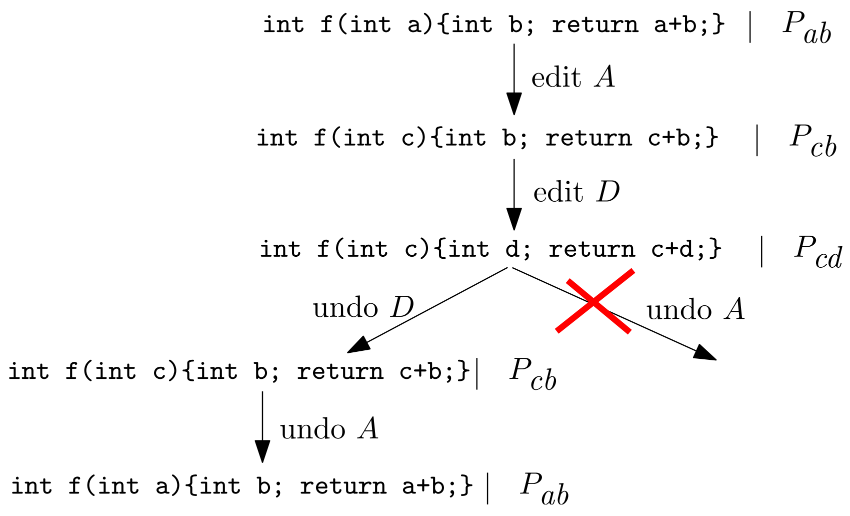

int f(int a){int b; return a+b;} | P_{ab}
edit A
int f(int c){int b; return c+b;} | P_{cb}
edit D
int f(int c){int d; return c+d;} | P_{cd}
undo D
undo A
int f(int c){int b; return c+b;} | P_{cb}
undo A
int f(int a){int b; return a+b;} | P_{ab}

Figure 2.4: Undo in the LEM for scenario 1.

can still choose to undo the earlier edit B', redo the earlier edit A, and keep the later edit C because they are independent of each other. Finally, we end up with the program P_{c5}. We capture this selective redo in Figure 2.6.

2.3 Entangled edits and partial selective undo

In the previous section, we observed that selective undo improves the LEM in that it provides a more flexible support for undoing edits. More specifically, as long as two edits are independent, either one can be undone without changing the other edit. Now how about the case that two edits are actually entangled, for example, the edits A and B in the scenario 2.

To see how selective undo behaves for the scenario 2, Figure 2.7 presents the attempt of undoing A directly after the edit B. However, it fails since the edit B depends on the edit A in that B changes the parameter that was previously renamed by A.

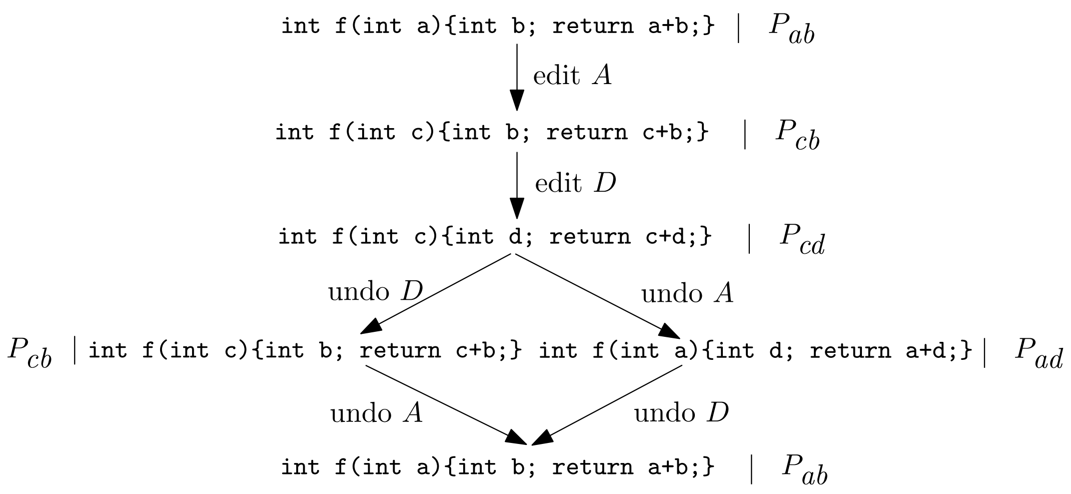

int f(int a){int b; return a+b;} | P_{ab}
edit A
int f(int c){int b; return c+b;} | P_{cb}
edit D
int f(int c){int d; return c+d;} | P_{cd}
undo D
undo A
P_{cb} | int f(int c){int b; return c+b;}
int f(int a){int d; return a+d;} | P_{ad}
undo A
undo D
int f(int a){int b; return a+b;} | P_{ab}

Figure 2.5: Selective undo for scenario 1.

However, it's not the case that the whole B depends on A. In fact, we can split B into two edits B_1 and B_2, where B_1 renames the parameter `c` to `d` and B_2 renames the local variable `b` to `c`. Thus, although B_1 depends on A, B_2 is independent of A. Figure 2.8 presents this split and the resulting programs. The figure makes it clear that B_2 in fact depends on A. In the figure, we use dashed arrows to denote the applications of edits conceptually. For example, while A is applied to the original program P_{ab}, we can envision that it is applied to the program P_{ac} since A changes the parameter `a`, which still exists in P_{ac}.

With this view, it should be clear that undoing A should be possible. Meanwhile, B_1 should be undone while B_2 can be kept. Figure 2.8 also demonstrates the process of applying this undo operation. In addition to selective undo, this operation relies on the notion of *partiality* of edits. Specifically, in order to undo A, we have partitioned B into B_1 and B_2. We call this undo operation *partial selective undo*.

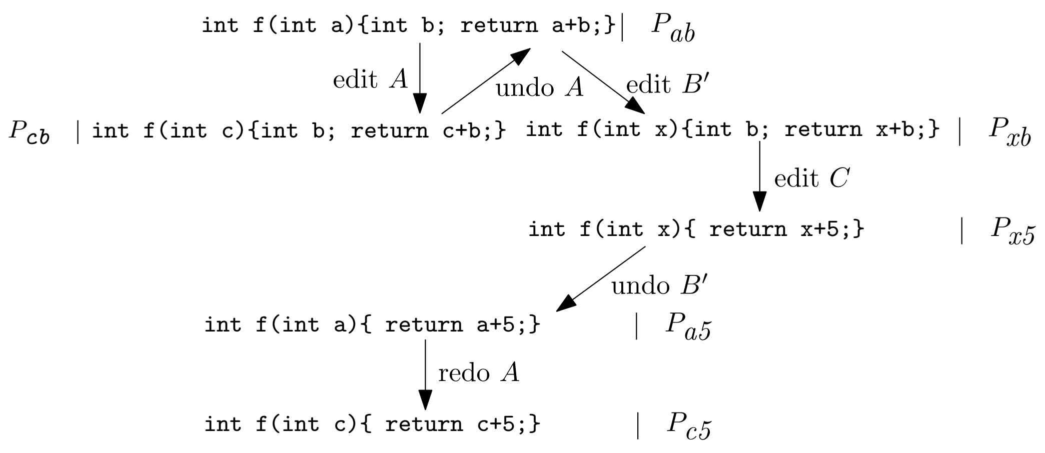

int f(int a){int b; return a+b;} | P_{ab}
edit A
undo A
edit B'
P_{cb} | int f(int c){int b; return c+b;}
int f(int x){int b; return x+b;} | P_{xb}
edit C
int f(int x){ return x+5;} | P_{x5}
undo B'
int f(int a){ return a+5;} | P_{a5}
redo A
int f(int c){ return c+5;} | P_{c5}

Figure 2.6: Selective redo for scenario 3.

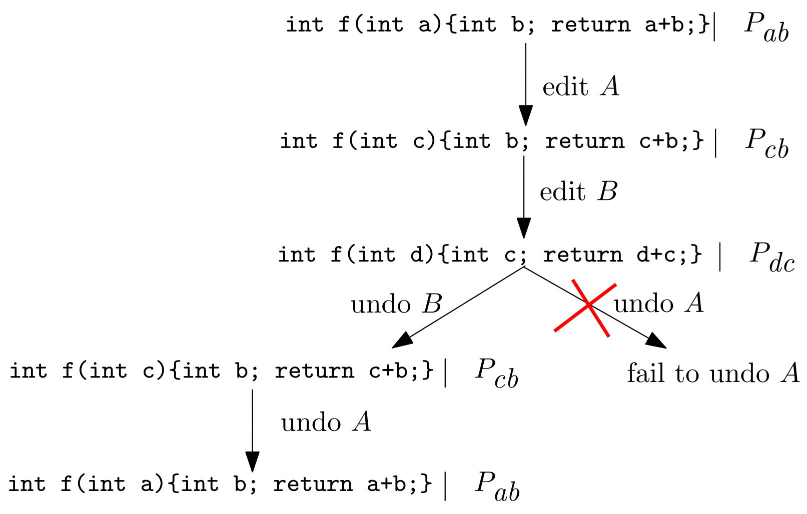

int f(int a){int b; return a+b;} | P_{ab}
edit A
int f(int c){int b; return c+b;} | P_{cb}
edit B
int f(int d){int c; return d+c;} | P_{dc}
undo B
undo A
int f(int c){int b; return c+b;} | P_{cb}
fail to undo A
undo A
int f(int a){int b; return a+b;} | P_{ab}

Figure 2.7: Selective undo fails for the scenario 2.

From Figures 2.2 and 2.7, we know that the LEM and selective undo can produce three programs P_{ab}, P_{cb}, and P_{dc} in scenario 2, which are produced by not applying any edit, applying the edit A only, and applying both the edits A and B, respectively. With the partial selective undo, we can reach an additional program P_{ac} as shown in Figure 2.8.

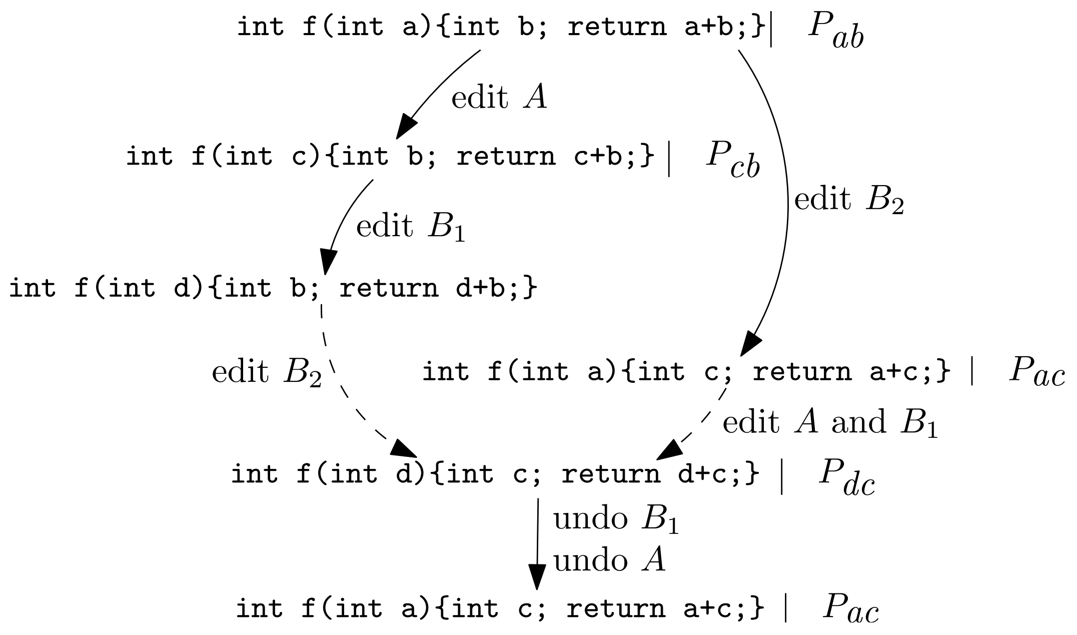

int f(int a){int b; return a+b;} | P_{ab}
edit A
int f(int c){int b; return c+b;} | P_{cb}
edit B_2
edit B_1
int f(int d){int b; return d+b;}
edit B_2
int f(int a){int c; return a+c;} | P_{ac}
edit A and B_1
int f(int d){int c; return d+c;} | P_{dc}
undo B_1
undo A
int f(int a){int c; return a+c;} | P_{ac}

Figure 2.8: Undo in partial selective undo for scenario 2.

While selective undo is more flexible than the LEM, partial selective undo is yet more flexible in that it allows us to reach more editing states, for example, the program P_{ac} in scenario 2. Although discovering the hidden program P_{ac} can simplify the editing process, it is not easy to achieve this without a good tool support. This project report presents a method to discover, represent, and exploit such hidden programs with support for partial selective undo.

At this time, we find that without a formal representation, it takes a lot of effort to keep track of all the edits, remember the relations among these edits, and reason about the resulting programs. We will go back to these examples in Chapter 3 after we introduce a variation representation in the next section.

2.4 Variation representation

The choice calculus provides a systematic and formal representation of software variations. It has been widely explored in software variation research, for example, in variation programming [Erwig and Walkingshaw, 2013], variational typing [Chen et al., 2012, 2014], projectional editing [Walkingshaw and Ostermann, 2014], type checking software product lines [Kästner et al., 2012], and other analyses of software product lines. In this project report, to represent program editing evolutions, we mainly utilize its choice representations and selection semantics. We focus our discussion on binary choices. Each choice consists of a *dimension name* and two *alternatives*. The following example displays a general form on how we use choices to represent program edits.

$$vp : \texttt{int}\ A\langle \texttt{a}, \texttt{b}\rangle \texttt{=1;}$$

Here *vp* is a variational program, which declares the variable `a` or `b` and assigns the value `1` to it. The tag A is called a *dimension*. Variables `a` and `b` are the left and right alternatives of choice A, respectively. Choices under the same dimension represent one program edit, and choices under different dimensions represent independent edits. In general, a variational program may have a more complicated variational structure, for example, nesting a variational expression in a choice. Note that we use single letters in alphabetical order to show the edits in the order in which they occurred. Thus choice B is an edit that happened later than A.

The syntax of variational programs is given as follows.

$$e \ ::= \ token \mid D\langle e, e\rangle \mid e \ldots e$$

This syntax says that a variational program can be a token, a choice of variational programs, or a concatenation of other programs. We can represent the variational program *vp* as follows.

$$\begin{aligned}
&vp : e_1\ A\langle e_2, e_3\rangle\ e_4\\
&e_1 : \texttt{int}\\
&e_2 : \texttt{a}\\
&e_3 : \texttt{b}\\
&e_4 : \texttt{=1;}
\end{aligned}$$

An important operation on choice expressions is the selection of an alternative of some choice. To formally discuss this operation, we introduce the notion of selectors, which are defined as follows. Given a dimension D, we can form two selectors $\overline{D}$ and $\underline{D}$, corresponding to the left and right alternative of D, respectively. We use s to range over selectors. We can now formally define the selection operation $\lhd : e \times s \to e$ as follows. In the definition, we assume that there is no nesting of choices with the same name. This can always be achieved by applying choice simplification rules defined in [Erwig and Walkingshaw, 2011]. For example, the expression $A\langle 1, A\langle 2,3\rangle\rangle$ can be simplified to $A\langle 1,3\rangle$.

$$\begin{aligned}
token \lhd s &= token\\
D\langle e_1, e_2\rangle \lhd \overline{D} &= e_1\\
D\langle e_1, e_2\rangle \lhd \underline{D} &= e_2\\
D\langle e_1, e_2\rangle \lhd s &= D\langle e_1 \lhd s, e_2 \lhd s\rangle && \text{where } s \neq \overline{D} \wedge s \neq \underline{D}\\
(e_1 \ldots e_n) \lhd s &= (e_1 \lhd s) \ldots (e_n \lhd s)
\end{aligned}$$

In this project report, we want to produce plain expressions, which are expressions without any choices. We define the operation $\blacktriangleleft$ of the type $e \times \mathscr{S} \to e$ to realize this goal.

The variation space $\mathscr{S}$ over a set of dimension Δ is defined as follows.

$$\mathscr{S} = \bigtimes_{D \in \Delta} \{\overline{D}, \underline{D}\}$$

The operation $\blacktriangleleft$ is then defined as follows.

$$e \blacktriangleleft \{\} = e$$
$$e \blacktriangleleft \{s_1, s_2, \ldots, s_n\} = (e \lhd s_1) \blacktriangleleft \{s_2, \ldots, s_n\}$$

In the following, we call a set of selectors a *decision*, and we use δ to range over decisions. We observe that the second argument of $\blacktriangleleft$ is a set. Thus, for $\blacktriangleleft$ to be well defined, we need to ensure that the ordering of applying selections doesn't matter, that is,

$$e \lhd s_1 \lhd s_2 = e \lhd s_2 \lhd s_1$$

While we will not prove this result in general, we show it with one example in the following.

$$(A\langle B\langle a,b\rangle, D\langle c,d\rangle\rangle \lhd \underline{A}) \lhd \overline{D} = D\langle c,d\rangle \lhd \overline{D} = c$$
$$(A\langle B\langle a,b\rangle, D\langle c,d\rangle\rangle \lhd \overline{D}) \lhd \underline{A} = A\langle B\langle a,b\rangle, c\rangle \lhd \underline{A} = c$$

While choices are scoped with dimensions in the original work of the choice calculus [Erwig and Walkingshaw, 2011], we assume that all choices are globally scoped. This simplified view was also adopted in [Chen et al., 2012, 2014]. However, choice names are still significant in that choices with the same name are synchronized and choices with different names are independent.

Chapter 3: Choice Edit Model: Uncovering hidden relations

Based on the choice syntax and selection semantics in Section 2.4, we will introduce a non-linear edit model, which we call *choice edit model* (CEM), in this chapter. We use the latter two examples introduced in Section 2.1 throughout this chapter. Using the CEM, we are able to discover more program variants than using a linear edit model (LEM). Also we have a clearer view of program evolution for the program editing process. These observations are described in Section 3.2. In Section 3.3, we discuss two relationships among edits: contingent and divergent edits.

3.1 Edits as choices

The basic idea of the CEM is viewing edits as choices, which means that we use choices to express editing results. The question is how can the choice representation precisely distinguish different edits and efficiently navigate through different program variants? In this section, we use the examples introduced in Section 2.1 to illustrate our answer to this question.

Consider the first program edit of the scenario 2, which introduces the choice A and produces the variational program V_A as follows.

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

V_A

For this single choice A, selecting with the selector $\underline{A}$ applies the program edit A, which produces P_{cb}. Instead, choosing the selector $\overline{A}$ means not applying or reversing the previous edit A (undoing edit A), which produces the original program P_{ab}. Their selection results are shown below:

$$P_{cb} = V_A \blacktriangleleft \underline{A}$$
$$P_{ab} = V_A \blacktriangleleft \overline{A}$$

Now consider both program edits, which introduce both the choices A and B and produce the variational program V_{AB}, shown in the following.

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

V_{AB}

Similarly, applying only the edit A and not applying the edit B leads to the program P_{cb}. Applying both edits results in the new program P_{dc}. The selection results are presented below, which exactly correspond to the three cases in the linear edit model (LEM).

$$P_{ab} = V_{AB} \blacktriangleleft \{\overline{A},\overline{B}\}$$
$$P_{cb} = V_{AB} \blacktriangleleft \{\underline{A},\overline{B}\}$$
$$P_{dc} = V_{AB} \blacktriangleleft \{\underline{A},\underline{B}\}$$

So far we have considered selecting the variational program V_{AB} with the selectors $\{\overline{A},\overline{B}\}$, $\{\underline{A},\overline{B}\}$, and $\{\underline{A},\underline{B}\}$. But how about $\{\overline{A},\underline{B}\}$? Does this selection correspond to a possible program variant? The answer is yes. Applying the selection operation with $\{\overline{A},\underline{B}\}$, we obtain the following

$$P_{ac} = V_{AB} \blacktriangleleft \{\overline{A},\underline{B}\}$$

which is a reasonable program variant, although it cannot be reached by the LEM. The question is then what is the semantics of applying this selection? The literal meaning is not applying edit A but applying edit B. Although this seems to contradict the choice dependence since part of edit B depends on the edit A, the application of this selection can be interpreted as a partial selective undo. More specifically, it corresponds to reversing the previously applied edit A (undo A) but keeping the edit B. Since part of the edit B is independent of the edit A, undoing edit A will automatically undo only part of the edit B that depends on the edit A.

While choice expressions can have arbitrary variation structures, we impose some conditions of using them to represent program edits. First, for any given two dimensions D_1 and D_2, if some choices of D_2 appear in the left alternatives of some choices of D_1, then no choices of D_2 appear in the right alternative of any choice of dimension D_1. This is because programmers can take editing actions only on one particular program variant, which includes the same alternatives of the choices with the same names. That variant can be the original program or the result of some selections. This implies that the choices created by later edits can only appear in one of the two alternatives (or not at all) for each choice created by earlier edits.

Second, as mentioned before, edits are ordered, which means that if the choice C_2 occurs in choice C_1, then C_1 comes before C_2. Thus, choosing choice names in alphabetical order, the expression $A\langle \mathtt{a}, B\langle \mathtt{c}, \mathtt{d}\rangle\rangle$ means that the program produced by applying the edit A is transformed or changed to a new program by applying the edit B.

So far we have seen only expressions where edits have a chain structure, which means that the inner choice nests in the right alternative of the outer choice. For example, in the program V_{AB}, the choice B is in the right alternative of the choice A. The scenario 3 discussed in Section 2.1 generates the following variational program V_{ABC}, which has a branch structure. In this case, the inner choice (B) is the left alternative of the outer choice (A).

```
int f(int A⟨B⟨a,x⟩,c⟩) {
    C⟨int b;,ε⟩
    return A⟨B⟨a,x⟩,c⟩+C⟨b,5⟩;
}
```

V_{ABC}

The difference between the CEM and the linear model is that the CEM relaxes the restrictions of temporal ordering in the LEM. In the LEM, any two adjacent edits are ordered such that a later edit can be applied only when an earlier edit has been applied. Similarly, an earlier edit can be undone only when a later edit has been undone. In the CEM, this ordering constraint is weakened.

Although the choice representation contains all the information for undo and redo operations, we don't show it to the user directly for two reasons. First, as the number of choices is linear in the number of edits, showing the whole expression will create a huge cognitive burden. Second, showing the whole expression can also confuse users who might want to figure out which variant is the one they care about. Instead, we only show the variant determined by the decision reflecting the editing history. We will return to this in the next section.

3.2 Discovering program families

In both, the CEM and the LEM, program editing leads to a series of similar programs although only the latest version is shown. However, without a formal representation, we can hardly express the transformations and differences among such a huge amount of programs in a compressed and efficient way. Thanks to the choice representation, we can encode all versions of programs in one variational program.

I now explain how edits, edit operations, the choice expression, view decisions, and variants are related. A *view decision* is a decision as introduced in Section 2.4 that defines the current variant (called view) that a programmer sees when making edits. Since programmers shouldn't see and manipulate choice expressions directly, we need a connection between the choice expression that represents an editing history and the content on which the programmer performs editing operations. This connection is realized by a view decision. In particular, we use the current view decision to refer to the view decision that's used to select the choice expression to get the current content in the editor. If the choice D is created in response to an edit, then $\underline{D}$ is added to the current view decision. If the edit D is undone, then the selector $\underline{D}$ is changed to $\overline{D}$ in the current view decision. Dually, if the edit D is redone, the selector $\overline{D}$ is changed to $\underline{D}$ in the current view decision. Making an edit creates a choice in the currently "active" variant, which is obtained by selecting the choice expression with the current view decision. The selection operation is defined in Section 2.4. Choice expressions are created once edits have been made. Undo and redo edit operations will change the view decision. The following table lists the relationships between program edits, variational programs, the decisions chosen, and their corresponding variants for the undo example introduced in Section 2.2. Each row includes five columns, the current time point, the current edit, the generated variational program, the accumulated decision so far, and the visible variant produced by applying the current decision to the variational program in the same row.

Time	*Edit*	*Variational Program* (V)	*View Decision* (δ)	*Visible Variant* (P)
		P_{ab}	$\varnothing$	P_{ab}
1	A	V_A	$\{\underline{A}\}$	P_{cb}
2	B	V_{AB}	$\{\underline{A},\underline{B}\}$	P_{dc}
3	*undo A*	V_{AB}	$\{\overline{A},\underline{B}\}$	P_{ac}

From this table we observe that more program variants can be produced in the CEM than those in the LEM. The reason is that the CEM treats the portion of the B edit that doesn't depend on A as a partial, independent edit. We call this partial edit B^I. This separation allows us to reverse the edit A but preserve the edit B^I. This view is reasonable since there is no dependency between the edits A and B^I. In contrast, the variant corresponding to this case is exactly what is ignored by the LEM, which prevents the edit A from being undone before undoing the edit B.

We can observe that, in each row, the generated program variant can be obtained by selecting the corresponding variational program with the decision in that row. We formally define this relation as follows.

$$P = V \blacktriangleleft \delta$$

Moreover, the current view decision δ_i can be computed by merging the previous view decision δ_{i-1} with the present edit D expressed in the following equations.

$$\begin{aligned}
\delta_i &= \delta_{i-1} \cup \{D\} && \text{if } \textit{Edit} = D \\
\delta_i &= \delta_{i-1} - \{D\} \cup \{\overline{D}\} && \text{if } \textit{Edit} = \textit{undo } D \\
\delta_i &= \delta_{i-1} - \{\overline{D}\} \cup \{D\} && \text{if } \textit{Edit} = \textit{redo } D
\end{aligned}$$

With these relations, we can compute the results of the corresponding edits for the redo example in Section 2.2. The details are shown in the following table.

Time	Edit	*Variational Program* (V)	*View Decision* (δ)	*Visible Variant* (P)
		P_{ab}	$\varnothing$	P_{ab}
1	*A*	V_A	$\{\underline{A}\}$	P_{cb}
2	*undo A*	V_A	$\{\overline{A}\}$	P_{ab}
3	*B*	V_B	$\{\overline{A},\underline{B}\}$	P_{xb}
4	*C*	V_{ABC}	$\{\overline{A},\underline{B},\underline{C}\}$	P_{x5}
5	*undo B*	V_{ABC}	$\{\overline{A},\overline{B},\underline{C}\}$	P_{a5}
6	*redo A*	V_{ABC}	$\{\underline{A},\overline{B},\underline{C}\}$	P_{c5}

Note that the variational program V_B at the time 3 and the visible variant P_{a5} at the time 5 appear the first time, and we present them below.

```
int f(int A⟨B⟨a,x⟩,c⟩) {
    int b;
    return A⟨B⟨a,x⟩,c⟩+b;
}
```
V_B

```
int f(int a){ return a+5;}
```
P_{a5}

Conceptually, once the edits are applied, the corresponding choices will be added to the variational program and never be removed. Specifically, succeeding operations, such as undo and redo, do not eliminate choices in the variational program. However, these operations do change the present view decisions and program variants that are visible to the programmers.

3.3 Contingent and divergent edits

Each choice has two alternatives: the left alternative and the right alternative. Correspondingly, we have two kinds of relationships between edits: divergent edits and contingent edits. A divergent edit corresponds to the case of undoing a certain edit before applying a new edit, and a contingent edit represents the case that a later edit changes the result of a prior edit. The edit B is a divergent edit or a contingent edit of the edit A if B is nested in the left alternative or the right alternative of A, respectively. Consider the following choice expression e_1.

$$e_1 = A\langle B\langle e_1, e_2\rangle, C\langle e_3, e_4\rangle\rangle$$

Since B is in the left alternative of A, it is a divergent edit of A. Likewise, since C is in the right alternative of A, it is a contingent edit on A.

To formally discuss the relations among edits, we introduce the notion of dominant sets and contingent edits. A dominant set is a set of dimensions. We use Δ to range over dominant sets. We say that Δ is a *dominant* set of the contingent edit D in e if the effect of (part of) the edit D is visible only after all the edits in Δ have been applied. We use the judgement $e \vdash \Delta \hookrightarrow D$ to express the relation that the edit D is contingent on Δ.

In the expression e_1, we have $e_1 \vdash \{A\} \hookrightarrow C$, meaning that $\{A\}$ is a dominant set of the contingent edit C. For example, suppose we apply both edits A and C, and then undo

$$\textsc{Top} \quad D\langle e_1, e_2\rangle \vdash \{\} \hookrightarrow D \qquad \frac{e_1 \vdash \Delta_1 \hookrightarrow D \ \cdots \ e_n \vdash \Delta_n \hookrightarrow D}{e^* \vdash \Delta_1, \ldots, \Delta_n \hookrightarrow D}\ \textsc{Group}$$

$$\frac{e_\ell \vdash \Delta \hookrightarrow D}{C\langle e_\ell, e_r\rangle \vdash \Delta \hookrightarrow D}\ \textsc{Branch} \qquad \frac{e_r \vdash \Delta \hookrightarrow D}{C\langle e_\ell, e_r\rangle \vdash \Delta \cup \{C\} \hookrightarrow D}\ \textsc{Chain}$$

Figure 3.1: Inference rules for computing contingent edits.

both of them. Now although we can redo the edit C before redoing the edit A, the effect of the edit C is invisible until being triggered by applying the edit A. We say the edit C is in a pending state before having applied the edit A. We define a set of inference rules to compute this dependency relation.

For choice expressions, we give the inference rules for computing dominant sets and contingent edits in Figure 3.1.

The rule TOP initializes the dominant set for each edit to empty. Both the rules BRANCH and CHAIN recursively deliver the dominant sets of a particular edit D from the inner layer to the outer layer for two different choice structures. Specifically, the rule BRANCH keeps the dominant sets of a particular edit D if edit D is in the left alternative of choice C. The rule CHAIN computes the dominant sets of a particular edit D by adding C to each dominant set if the edit D is in the right alternative of choice C. As we illustrated in Section 3.1, an edit D can only occur in either the left or the right alternative of a choice C, but never in both. Thus only one of the rules BRANCH and CHAIN may apply. The rule GROUP clusters all the dominant sets for one particular edit D for the concatenated expression.

We can capture the essential edit structure of the undo example in the following simplified expression.

$$e_{AB} = A\langle \texttt{a}, B\langle \texttt{b}, \texttt{c}\rangle\rangle\ B\langle \texttt{d}, \texttt{e}\rangle$$

With the inference rules in Figure 3.1, we can compute dominant sets for the edits A and B, respectively by drawing the following derivation trees.

$$\textsc{Group}\ \dfrac{\textsc{Chain}\ \dfrac{B\langle \texttt{b}, \texttt{c}\rangle \vdash \{\}\hookrightarrow B\ \textsc{Top}}{A\langle \texttt{a}, B\langle \texttt{b}, \texttt{c}\rangle\rangle \vdash \{A\}\hookrightarrow B} \qquad B\langle \texttt{d}, \texttt{e}\rangle \vdash \{\}\hookrightarrow B\ \textsc{Top}}{A\langle \texttt{a}, B\langle \texttt{b}, \texttt{c}\rangle\rangle\ B\langle \texttt{d}, \texttt{e}\rangle \vdash \{\}, \{A\}\hookrightarrow B}$$

$$\textsc{Group}\ \dfrac{A\langle \texttt{a}, B\langle \texttt{b}, \texttt{c}\rangle\rangle \vdash \{\}\hookrightarrow A\ \textsc{Top}}{A\langle \texttt{a}, B\langle \texttt{b}, \texttt{c}\rangle\rangle\ B\langle \texttt{d}, \texttt{e}\rangle \vdash \{\}\hookrightarrow A}$$

In short, we have the following judgements about contingent edits for the expression e_{AB}.

$$e_{AB} \vdash \{\}, \{A\}\hookrightarrow B$$

$$e_{AB} \vdash \{\}\hookrightarrow A$$

The first judgement conveys that the edit B is only partially contingent on edit A because of the existence of $V_{AB} \vdash \{\}\hookrightarrow B$, which means that we can apply the partial edit B without applying any other edits. From the second judgement, we know that A is an independent edit. These results are consistent with our view on the CEM (Section 3.1). Similarly, we can repeat this process for the redo example. For this example, the dominant set for edits A, B, and C are empty sets, which means that the edits A, B and C are independent of each other.

Note that when applying the rule Group, there may exist some duplicate sets or overlapping subsets. We can certainly remove repetitive sets, but not overlapping subsets because that will cause the loss and incompleteness of the dependencies. For example, consider the relation $\{A\},\{A,B\}\hookrightarrow C$. One may be tempted to delete the edit A in the dominant set $\{A,B\}$ to avoid redundancy and obtain the reduced relation $\{A\},\{B\}\hookrightarrow C$. However, the reduced relation means that part of the edit C solely relies on the edit A, and the other part of the edit C solely depends on the edit B. This, in turn, means that selecting with the decision $\{\overline{A},\underline{B}\}$ will cause part of the edit C being applied. However, $\{A\},\{A,B\}\hookrightarrow C$ tells us that no part of C is triggered until at least A has been applied.

With the help of rules in Figure 3.1, we can prove that the CEM supports selective undo. Given an expression e and two edits A and B, if there is no dominant set Δ such that $A \in \Delta$ and $e \vdash \Delta \hookrightarrow B$, then there must be a subexpression e_1 in e that contains itself two subexpressions e_2 and e_3 and A occurs in e_2 and B occurs in e_3. In other words, edits A and B must involve different subexpressions of e. Thus, either edit can be undone without affecting the other. Therefore, we say that the CEM supports selective undo. Similarly, given an expression e and two edits A and B, if there is a dominant set Δ such that $A \in \Delta$ and $e \vdash \Delta,\{\}\hookrightarrow B$, we can derive following information about e. First, there must be a subexpression e_1 in e and e_1 contains two subexpressions e_2 and e_3. More specifically, there exist e', e'', and e''' such that $e_1 = e' e_2 e'' e_3 e'''$. Second, without the loss of generality, we assume that some part of the edit B occurs in e_2 and the other part of B is nested in A in e_3. Thus, undoing the edit A will undo the edit B in e_2, but not the edit B in e_3. Therefore, we conclude that the CEM supports partial selective undo.

$$\textsc{Plain}\quad token \curlyvee \{\} \qquad \textsc{P-P}\quad D\langle token, token\rangle \curlyvee \{\} \qquad \textsc{Diverge}\quad C\langle e_\ell D\langle e_1, e_2\rangle e_r, e_3\rangle \curlyvee \{C, D\}$$

$$\textsc{Group}\quad \frac{e_1 \curlyvee \Delta_1 \quad \cdots \quad e_n \curlyvee \Delta_n}{e^* \curlyvee \Delta^*} \qquad \textsc{Choice}\quad \frac{e_\ell \curlyvee \Delta_\ell^* \qquad e_r \curlyvee \Delta_r^*}{C\langle e_\ell, e_r\rangle \curlyvee \Delta_\ell^* \cup \{C\}, \Delta_r^*}$$

Figure 3.2: Inference rules for computing divergent edits.

We employ the notation $A \curlyvee B$ to express that edits A and B are divergent, which corresponds to the case when choice B is nested in the left alternative of choice A. Note that it's impossible for A to nest in B due to the naming convention. We say the edit B *branches* from the edit A, or the edits A and B are *divergent* from each other. We write $e \curlyvee \Delta$ to denote that all edits in Δ are mutually divergent. We write Δ^* for a list of Δs starting with Δ_1 and ending with Δ_n. Figure 3.2 defines the rules for computing divergent edits.

The first two rules indicate that there are no divergent edits in an expression without choice nestings. The rule DIVERGE gives the base case for two divergent edits. The rule GROUP collects all the divergent sets for concatenated expressions. The rule CHOICE adds the dimension C to all divergent sets obtained from the left alternative. It also collects the divergent sets from the right alternative.

We use the following expression e_{ABCD} to show how to apply the rules in Figure 3.2.

$$e_{ABCD} = A\langle B\langle C\langle \texttt{a}, \texttt{b}\rangle, \texttt{c}\rangle D\langle \texttt{d}, \texttt{e}\rangle, \texttt{f}\rangle$$

We can compute its divergent edits by drawing the following derivation trees. The result is two divergent sets $\curlyvee\{A,B,C\}$ and $\curlyvee\{A,D\}$.

$$\textsc{Choice}\ \dfrac{\textsc{Group}\ \dfrac{\textsc{Diverge}\ B\langle C\langle \texttt{a},\texttt{b}\rangle,\texttt{c}\rangle \curlyvee \{B,C\} \qquad D\langle \texttt{d},\texttt{e}\rangle \curlyvee \{\}\ \textsc{P-P}}{B\langle C\langle \texttt{a},\texttt{b}\rangle,\texttt{c}\rangle D\langle \texttt{d},\texttt{e}\rangle \curlyvee \{B,C\},\{\}} \qquad \texttt{f} \curlyvee \{\}\ \textsc{Plain}}{A\langle B\langle C\langle \texttt{a},\texttt{b}\rangle,\texttt{c}\rangle D\langle \texttt{d},\texttt{e}\rangle,\texttt{f}\rangle \curlyvee \{A,B,C\},\{\}}$$

$$\textsc{Diverge}\ A\langle B\langle C\langle \texttt{a},\texttt{b}\rangle,\texttt{c}\rangle D\langle \texttt{d},\texttt{e}\rangle,\texttt{f}\rangle \curlyvee \{A,D\}$$

With the same process, we find that no divergent sets exist in the undo example. As for the redo example, there is one divergent set $e_{ABC} \curlyvee \{A,B\}$.

A choice expression can contain contingent edits as well as divergent edits. For example, consider the choice expression e_{ABC}.

$$e_{ABC} = A\langle \texttt{a}, B\langle C\langle \texttt{b},\texttt{c}\rangle,\texttt{d}\rangle\rangle B\langle \texttt{e},\texttt{f}\rangle C\langle \texttt{g},\texttt{h}\rangle$$

We have the following judgements about dependencies among edits for the expression e_{ABC}.

$$e_{ABC} \vdash \{\},\{A\} \hookrightarrow B$$

$$e_{ABC} \vdash \{\},\{A\} \hookrightarrow C$$

$$e_{ABC} \vdash \{\} \hookrightarrow A$$

$$e_{ABC} \curlyvee \{B,C\}$$

These two kinds of edits can be entangled with each other, making the choice expressions hard to comprehend. Program edit graphs (PEGs), a visualization representation, are better at revealing the relations among the choice expression, applying edits, and the corresponding resulting program variant. We describe PEGs in Chapter 4.

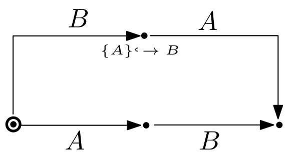
Chapter 4: Program Edit Graphs

Choice expressions encode editing histories in a compact way. The CEM that is based on choice representations can discover hidden editing states and discover relations between edits, which are essential for supporting partial selective undo and redo. However, working directly with choice expressions should be avoided for many reasons. First, presenting choices to users directly increases the cognitive burden because they have to understand choice constructs and the related conventions. Second, editing choice expressions directly may introduce, create, and alter choice structures that could violate the invariants of the choice representation.

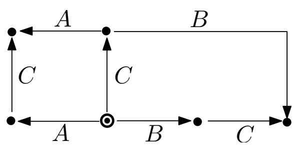
This chapter presents Program Edit Graphs (PEGs), a graphical notation for representing the relations among variational programs, applying edits, and the resulting variants. In Section 4.1, I present the principles of PEGs through two examples. I give a nondeterministic algorithm for constructing PEGs in Section 4.2. I observe that PEGs are quite dense. In Section 4.3, I discuss the implication of this observation and present an approach to reduce the complexity of PEGs.

4.1 Principles and examples

A PEG $G = (N, E, F)$ consists of three parts, a set of nodes N, a set of edges E, and a partial function F that maps nodes to contingent edit information. Each node $n \in N$ is a



(a) PEG for V_{AB}.



(b) PEG for V_{ABC}.

Figure 4.1: PEGs for scenarios two and three, respectively.

set of dimensions. Each edge $\eta \in E$ has the form (n_s, n_t, D), meaning that the directed edge starts from n_s and ends at n_t with the label D. The relation $n_t = n_s \cup D$ always holds. When $n \in dom(F)$, then $F(n)$ is a set of pairs. Each pair has the form (Δ^*, D) denoting that $e \vdash \Delta^* \hookrightarrow D$, where e is the expression for which the PEG is being constructed.

We further distinguish different kinds of nodes. We use two concentric circles to denote the unique source node of the PEG. A black dot without outgoing edges denotes a sink. Each PEG has only one source node, but may have several sinks. Generally, if the choice calculus expression has no branch structure, that is, no choices are nested inside the left alternative of other choices, then the corresponding PEG has one sink. However, if the expression contains branch structures, then its PEG must have more than one sink. For example, Figures 4.1a and 4.1b present the PEGs for the variational expressions V_{AB} and V_{ABC} introduced in Section 3.1, respectively. We observe that Figure 4.1a contains only one sink but Figure 4.1b contains two since B is nested in the left alternative of A in V_{ABC}.

In Figure 4.1a, $F = \{\{B\} \mapsto (\{A\}, B)\}$. This means that the edit B is, although applied, pending until the edit A has been applied. Another interpretation of $\{A\} \hookrightarrow B$ on the node $\{B\}$ is that, at the time when B is applied but A hasn't, only part of B has effect. As in the expression V_{AB}, part of B is nested in A but the other part isn't, which means that on the node $\{B\}$, the part outside A has been applied while the part inside A is pending and will be triggered after A has been applied.

Given an expression e and the corresponding PEG G, we can compute view decisions and the corresponding program variants and perform edits and undo as follows. First, we use Δ to denote all the dimensions included in e. The view decision for the node n can be computed as follows.

$$\delta = \{\overline{D} \mid D \in \Delta - n\} \cup \{\underline{D} \mid D \in n\}$$

The program variant P corresponds to that node can be obtained as follows.

$$P = e \blacktriangleleft \delta$$

Thus, the source node corresponds to the view decision that contains the left alternatives of all dimensions, denoting the variant in which no edits have been applied. Based on the relations among nodes, view decisions, and the corresponding variants, if (n_s, n_t, D) is an edge in the PEG, then the variant corresponding to n_t is obtained by applying the edit D to the variant corresponding to n_s. For example, in Figure 4.1a, the source node corresponds to the variant P_{ab}. Following the edge labeled with A, we arrive at the node that corresponds to the decision $\{\underline{A}, \overline{B}\}$. According to Section 3.1, $V_{AB} \blacktriangleleft \{\underline{A}, \overline{B}\} = P_{cb}$. Moreover, according to Figure 2.2, applying edit A to P_{ab} yields P_{cb}. Thus, PEGs truly model the editing process.

Interestingly, traversing an edge (n_s, n_t, D) backward means to undo the edit D. Moreover, by undoing D, the variant corresponding to n_s can be achieved from n_t. For example, the sink in Figure 4.1a corresponds to the variant P_{dc}. When we go backward the edge labeled B, we arrive at the variant P_{cb}. This matches the editing process shown in Figure 2.7. We observe that there is another edge labeled A ending in the sink. By undoing A, we end up with the node $\{B\}$, which leads to the view decision $\{\overline{A},\underline{B}\}$. Thus, we derive that the corresponding variant is P_{ac}. This result coincides with the editing process shown in Figure 2.8.

The ability to traverse edges backwards increases the expressiveness of PEGs. Given any node, we can follow any outgoing edge to move to another node, which corresponds to applying or redoing an edit. Meanwhile, we can follow any incoming edge to go backwards, which corresponds to undo the edit. Thus, the directions of the edges should not be understood as restricting editing flows but should be viewed as indicating the actions of undo or redo only. If we view PEGs as undirected graphs but remember their edge directions, then a random walk on a given PEG denotes an editing sequence consisting of applying edits, undo, and redo operations.

4.2 Construction

Given an expression, the idea of constructing the PEG is to start with the source node and then gradually add edges and nodes until no more edges can be added. Specifically, given the expression e, the construction process consists of two phases: the initialization phase and the iteration phase.

In the first phase, we initialize the PEG G_0 and the related variables as follows. Besides the following information, we maintain the global expression e so that it's accessible during the whole construction process.

$$\begin{aligned} e_0 &= e \\ D_0 &= \text{any top level choice in } e_0 \\ N_0 &= \{\varnothing\} \\ E_0 &= \varnothing \end{aligned}$$

In the second phase, we iterate through following steps until no progress can be made. Specifically, we compute the result for the $(i+1)$st iteration based on that for ith iteration as follows.

$$\begin{aligned} e_{i+1} &= e_i \blacktriangleleft \overline{D_i} \cdot e_i \blacktriangleleft \underline{D_i} \cdot D_i\langle \mathtt{a}, \mathtt{b} \rangle \\ D_{i+1} &= D \text{ where } e_{i+1} = e_p \cdot D\langle e_l, e_r \rangle \cdot e_q \\ n_{i+1} &= \mathit{pick}\ \{ n \in N_i \mid \mathit{extensible}(n, D) \} \\ n'_{i+1} &= n_{i+1} \cup \{ D_{i+1} \} \\ N_{i+1} &= N_i \cup \{ n'_{i+1} \} \\ E_{i+1} &= E_i \cup \{ (n_{i+1}, n'_{i+1}, D_{i+1}) \} \end{aligned}$$

The essential idea of each iteration is that based on the current expression e_{i+1}, a dimension D_{i+1} is chosen. The current expression is a concatenation of three parts. The first part is the left alternative from the dimension D_i of the previous expression e_i. This part ensures that the dimensions in the left alternative becomes available in subsequent iterations. The second part is similar but taking the right alternative from D_i and the purpose is to make sure that the dimensions in the right alternative become available in subsequent iterations. Similarly, the third part $D_i\langle \mathtt{a}, \mathtt{b} \rangle$ makes sure that D_i will be available in later iterations. Based on the current expression, a top-level dimension is chosen, and we use D_{i+1} to denote it. Moreover, an arbitrary node n_{i+1} in N_i may be picked as long as extending that node with D_{i+1} is valid. If such a node exists, then the node set N_{i+1} and edge set E_{i+1} are updated accordingly.

The most critical operation is picking a node in the previously constructed nodes such that an edge starting from that node and with label D_{i+1} may be added in this iteration. The relation *extensible*(n,D) is satisfied if all of the following conditions are met:

(a) $D \notin n$,

(b) $D \notin \Delta^*(n)$ for $e \curlyvee \Delta^*$, and

(c) there is some Δ such that $e \vdash \Delta \hookrightarrow D$ and $\Delta \subseteq n$.

The first condition ensures that no node will contain a certain dimension multiple times. In terms of editing, this means that no edit will be applied more than once to the same program. The second condition ensures that D is not divergent to any dimension in n. The operation $\Delta^*(n)$ is defined as follows.

$$\Delta^*(n) = \bigcup_{D \in n, 1 \le i \le m} \Delta_i(D)$$

$$\Delta(D) = \begin{cases} \Delta - \{D\} & \text{if } D \in \Delta \\ \varnothing & \text{otherwise} \end{cases}$$

Since $V_{ABC} \curlyvee \{A,B\}$, Figure 4.1b doesn't have an outgoing edge labeled with B from $\{A\}$ and $\{A,C\}$. Similarly, from $\{B\}$ and $\{B,C\}$, there is no outgoing edge labeled with A.

The third condition ensures that before applying D_{i+1}, all the edits that D_{i+1} depends on must have been applied. Intuitively, if D_{i+1} is nested in the right alternative of other choices, then the edits correspond to those choices must be applied before D_{i+1} may be applied. Of course, if D_{i+1} is a top-level dimension in e, then D_{i+1} can be applied at any

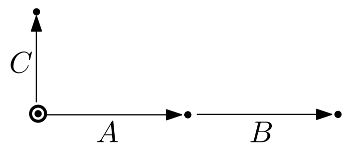


Figure 4.2: PEG for some expression that has both contingent and divergent edits.

time since $e \vdash \{\} \hookrightarrow D_{i+1}$. For example, since both A and B appear on the top level in V_{AB}, they may be applied at any time, as can be seen in Figure 4.1b. However, the situation is quite different in Figure 4.2, which shows the PEG for the expression $A\langle C\langle \mathtt{a},\mathtt{b}\rangle, B\langle \mathtt{c},\mathtt{d}\rangle\rangle$. We observe that B is nested in right alternative of A only, which is reflected in Figure 4.2 that B can appear on the edge coming out from the node $\{A\}$ only.

Thus, the whole construction process starts with the initialization and stops when no more edges can be added, which happens when no dimension exists such that *extensible*(n,D) holds for some n. After the construction is completed, we need to add F. Specifically, for each node n in the final PEG, we add $F(n)$ as follows. Here we assume that $\Delta^* - n$ is not empty. If $\Delta^* - n$ is empty for some D', we simply drop the item $(\{\},D')$.

$$F(n) = \{(\Delta^* - n, D) \mid D \in n, e \vdash \Delta^* \hookrightarrow D\}$$

Specifically, for each $D \in n$, we first derive its dominant sets Δ^* and then subtract n. The notation $\Delta^* - n$ is defined as follows.

$$\{\Delta - n \mid \Delta \in \Delta^*\}$$

In other words, $\Delta^* - n$ collects for each Δ in Δ^* the result of $\Delta - n$ if $\Delta - n$ is nonempty.

Since $V_{AB} \vdash \{A\} \hookrightarrow B$, the node $\{B\}$ in Figure 4.1a, the PEG for V_{AB} has $F(\{B\}) = \{(\{A\},B)\}$. For the node $\{A,B\}$, although we still have $V_{AB} \vdash \{A\} \hookrightarrow B$, we don't have an

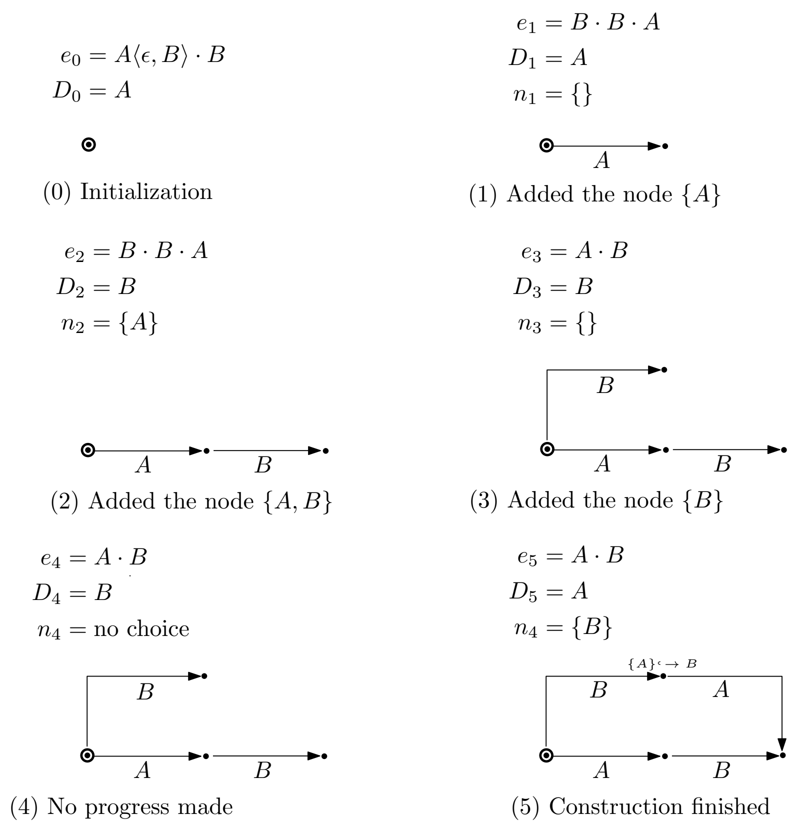
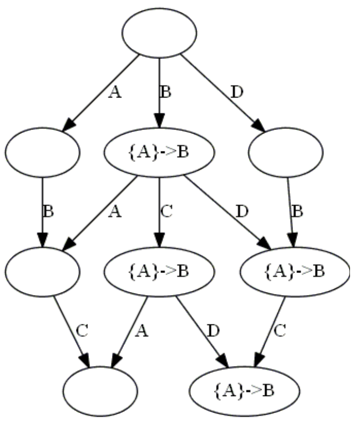


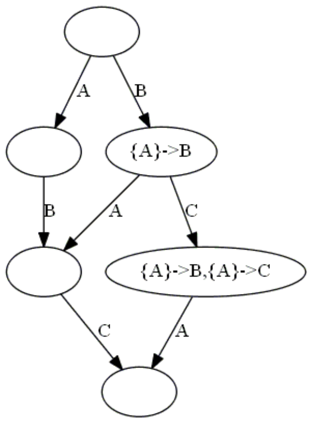
Figure 4.3: The process of constructing the PEG for V_{AB}.

entry for $\{A,B\}$ in F since $\{A\} - \{A,B\} = \varnothing$. Intuitively, both edits A and B have been applied in the node $\{A,B\}$, and thus no edit is pending.

Based on the construction process, Figure 4.3 depicts this process for the expression V_{AB}. Note that we have simplified the presentations of expressions by leaving out all *tokens*. For example, we write D for $D\langle token, token\rangle$ and $A\langle -, B\rangle$ for $A\langle token, B\langle token, token\rangle\rangle$. We use this convention for the rest of this chapter. In each step i, we list the expression e_i, the dimension chosen D_i, the node chosen n_i, and the PEG built so far. We observe that in some steps no progress can be made. For example, in step (4) when B is chosen as



(a) PEG for $A\langle D,B\rangle \cdot B\langle -,C\rangle$.



(b) PEG for $A\langle -,B\langle -,C\rangle\rangle \cdot B\langle -,C\rangle$.

Figure 4.4: PEGs produced by the prototype.

the dimension, no extensible node is available and thus no node or edge can be added. The process is completed with step (5) when the edge $(\{B\},\{A,B\},A)$ is added.

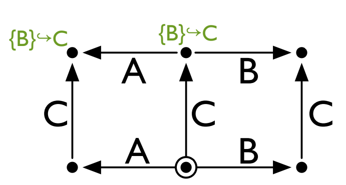
We have implemented a prototype for constructing PEGs in Haskell. The prototype also goes through several iterations. Although the algorithm presented here is nondeterministic, the prototype is deterministic. In particular, it maintains `fringe` for storing all nodes that maybe expanded later. The `fringe` is initialized to $\{\varnothing\}$, denoting that the source node is extensible. In each iteration, the first node in `fringe` is removed and is referred to as n. For each dimension D in the original expression, we check if $\mathit{extensible}(n,D)$ is satisfied. If this is the case, we further check if $n \cup \{D\}$ has been visited. If not, we add $n \cup \{D\}$ to the end of `fringe` and to the PEG node set. For each added node, we update F accordingly. We also add the edge $(n, n \cup \{D\}, D)$ to the PEG edge set. The iteration stops when `fringe` becomes empty. At that time, the final PEG is returned.

Figure 4.4 shows two PEGs produced by our prototype. In these figures, I have omitted node labels since they are very easy to construct. The label of each node is simply the set of all labels on the path from the source node to that node. For example, the bottom node in Figure 4.4b would have the label $\{A,B,C\}$. Instead, for each node n when $n \in dom(F)$ we show $F(n)$ for that node.

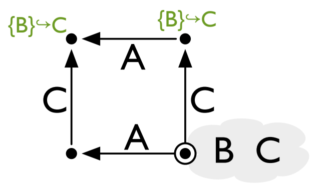
Some tricky questions may be addressed easily with the help of PEGs. For example, it is unclear what edits can be undone from the final state of the variational program $A\langle -,B\langle -,C\rangle\rangle \cdot B\langle -,C\rangle$. By looking at the corresponding PEG in Figure 4.4b, one immediately concludes that the edits A and C, but not B, can be undone. One may wonder why B couldn't be undone. The reason is that in both occurrences of B, C appears in the right alternative of B, which means C always depends on B. Thus, in the state when both B and C are applied, as in the final state $\{A,B,C\}$, B couldn't be undone until C has been undone.

4.3 Discussion

From all the PEGs presented in this chapter, in particular the PEGs in Figure 4.4, we observe that PEGs are quite dense. The question is, what is the implication of the density of PEGs, or what does the density measure? At the end of Section 4.1, we concluded that by viewing PEGs as undirected graphs, a random walk on the PEG represents a sequence of editing operations, including applying new edits and undoing and redoing previous operations. Thus, the fact that PEGs are dense reflects the fact that PEGs support flexible undo and redo operations.



(a) PEG for $A\langle B\langle -,C\rangle,-\rangle \cdot C$.



(b) Branch filter for $A \cdot C$.

Figure 4.5: PEG and branch filter.

Nevertheless, showing all the condensed information in a PEG might be overwhelming for the user. In particular, the PEG for a choice calculus expression with n independent dimensions contains 2^n nodes. When all dimensions are independent, any node containing any combination of dimensions is valid. When dimensions are dependent, the situation is more complicated. A precise complexity measurement is difficult to give since expressions may have arbitrary choice structures. However, we do observe that both chain and branch structures help to reduce the number of nodes. For example, due to the divergent relation $V_{ABC} \curlyvee \{A,B\}$, the PEG for V_{ABC} in Figure 4.1b contains 6 rather than 8 (2^3) nodes. Similarly, for $e_5 = A\langle -,B\langle -,C\rangle\rangle \cdot B\langle -,C\rangle$, we have $e_5 \vdash \{A,B\},\{B\}\hookrightarrow C$ and $e_5 \vdash \{A\}\hookrightarrow B$. Due to the contingent edits, the PEG for e_5 in Figure 4.4b contains 6 and not 8 nodes.

However, in general, the exponential relation between the number of the nodes and the number of the dimensions holds. One observation that can help here is that the user usually focuses on a small part of a big variational program. Based on that view, we can try to build the PEG for that part only and expand the PEG for other parts when required. We call this idea of showing the PEG for only certain branches *branch filters*.

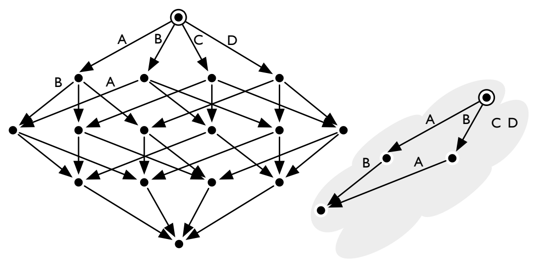


Figure 4.6: PEG for $A \cdot B \cdot C \cdot D$ (left) and branch filter for $A \cdot B$.

For example, Figure 4.5a shows the PEG for the expression $A\langle B\langle -,C\rangle,-\rangle \cdot C$. The branch filter for the branch $A \cdot C$ is shown in Figure 4.5b.

Although branch filters don't show the full PEG, the F component for the shown nodes contains complete contingent edits information. For example, while edges labeled with B are absent in Figure 4.5b, the nodes $\{C\}$ and $\{A,C\}$ still show $\{B\}\hookrightarrow C$, which indicate that the C edits are partial and the part that depends on B has no effect at this time. The use of branch filters can significantly reduces the number of nodes. For example, the branch filters in Figure 4.6 reduces the number of nodes from 16 to 4.

Although branch filters can help to reduce the complexity of the PEG for the whole variational program, it doesn't help for the branch being shown. In other words, the size of a branch filter is still exponential in the number of dimensions in that branch. Chapter 5 discusses this issue.

Chapter 5: Scalability

When the dimensions are independent, the size of a PEG is exponential in the number of the dimensions in the corresponding expression. Thus constructing and presenting PEGs immediately becomes intractable for expressions, except the most trivial ones. In particular, if we represent each edit as an independent dimension, then choice expressions can contain hundreds of dimensions. Section 4.3 presents the method of using branch filters to show just part of the PEG to the user. However, the part that is shown to the user may be still large. This chapter investigates another method of addressing this issue. The idea is to assign the same dimension to edits sharing the same intention, which reduces the number of dimensions significantly and the sizes of PEGs exponentially. Section 5.1 explains this idea in detail. In Section 5.2, I present some empirical results about the feasibility of this idea.

5.1 Grouping primitive edits into conceptual edits

In the previous section we have seen how branch filters can make PEGs more comprehensible by showing just part of them. However, branches can also be very big if they contain many edits. A potential approach to address this problem is to group different edits into a single edit so that they are assigned the same dimension. To see the significance of this idea, consider the following example. If we represent each edit in a unique dimension, then we may get the following expression.

$$A\langle 1,2\rangle \; + \; B\langle 3,4\rangle \; + \; C\langle 5,6\rangle \; + \; D\langle 7,8\rangle$$

However, if the first and last two edits belong together and are viewed as a single edit, then we get the following expression instead.

$$A\langle 1,2\rangle \; + \; A\langle 3,4\rangle \; + \; B\langle 5,6\rangle \; + \; B\langle 7,8\rangle$$

We observe that while the PEG for the first expression has 16 nodes, that for the second expression has only 4 nodes, reduced by a factor of 4.

In the following, we distinguish between different kinds of edits.

1. A *primitive edit* displays the differences at the lowest level, which can be a list of lines, a sequence of characters, and even a single character between two plain text files.

2. A *lexical edit* denotes a change of a meaningful atomic unit, which is known as a token. Tokens can be names, keywords, identifiers or any low level language elements.

3. A *syntactic edit* involves changes in higher-level program constructs, for example, expressions, statements, or functions.

4. A *conceptual edit* consists of many syntactic edits or lexical edits that serve the purpose of implementing a conceptual change. The term conceptual change has a broad meaning, it may denote an addition of a feature in the source node code within a single file, a refactoring of programs, or even a bug fix.

```
public int days(int month) {
    int [] l = {1,3,5,7,8,10,12};


    if (Arrays.asList(l).contains(month))
        return 31;
    else
        return 30;
}
```

```
public int days(int month) {
    int [] large = {1,3,5,7,8,10,12};
    int more = 31;
    int less = 30;
    if (Arrays.asList(large).contains(month))
        return more;
    else
        return less;
}
```

Figure 5.1: Two different implementations, old (left) and new (right), for returning the number of days for the given month.

```
Refactoring kind: RENAME_LOCAL_VARIABLE
Refactoring ID: 3
Arguments count: 2
Key: NewName
Value: large
Key: OldName
Value: l
Timestamp: 1406698679627
```

```
Refactoring kind: EXTRACT_LOCAL_VARIABLE
Refactoring ID: 4
Arguments count: 2
Key: ExtractedValue
Value: 31
Key: VariableName
Value: more
Timestamp: 1406698693783
```

Figure 5.2: Conceptual edits discovered by CodingTracker for the programs in Figure 5.1.

Conceptual edits are quite common in practice, in particular, when the initial development has been finished. There has been a lot of work of identifying conceptual edits from primitive edits. First, Dig et al. [2007] and Negara et al. [2012a; 2013] explored this idea from the perspective of program refactorings.

We use the example presented in Figure 5.1 to illustrate this idea. In the figure, we present two programs for implementing the same functionality of returning the number of days for the given month in Java. The program on the right is the new program, where underlined code highlights the changes. For this pair of programs, CodingTracker [Negara et al., 2012a, 2013] discovers three conceptual edits. The first conceptual edit is the refactoring `RENAME_LOCAL_VARIABLE`, presented in detail in the left of Figure 5.2. Since the local variable `l` has been renamed to `large` at two different places in the program, they are identified as one conceptual edit. Both the second and third edits are about the

```
-- The old version                          -- The new version

public class VerticalPlot{                  public class VerticalPlot{
    void draw(Graph g, Shape s){...} }          void draw(Graph g)... }
public class VerticalRenderer{              public class VerticalRenderer{
    void draw(Graph g, Shape s){...} }          void draw(Graph g, Shape s)... }
public class HorizontalPlot{                public class HorizontalPlot{
    void draw(Graph g, Shape s){...} }          void draw(Graph g){...} }
public calss HorizontalAxis{                public calss HorizontalAxis{
    public int height(){...} }                  public int getHeight(){...} }
public calss VerticalAxis{                  public calss VerticalAxis{
    public int height(){...} }                  public int getHeight(){...} }

-- Inferred rules

for all x:*.*(Graph, Shape)
          except VerticalRenderer.draw(Graph, Shape), argDelete(x, Shape)
for all x:*Axis.height() procedureRename(x, height, getHeight)
```

Figure 5.3: Two versions of a drawing editor (up) and the inferred edits (down).

refactoring `EXTRACT_LOCAL_VARIABLE`, and we present one edit on the right of Figure 5.2. This edit is about extracting the constant value `31` into the local variable `more`.

A very different approach of discovering conceptual edits is to use rule deduction to infer rules that describe low-level changes. A typical tool is LSdiff [Kim and Notkin, 2009; Nguyen et al., 2010; Kim et al., 2013]. Figure 5.3 presents two versions of a drawing tool and the conceptual edits discovered by LSdiff [Kim et al., 2013]. We observe that no refactorings can be identified from these edits because they involve different classes and have no close relation (except that they are in the same package). For these changes, LSdiff identifies two edits, represented in two rules. The first rule says that, in all the classes, all the methods that have parameter type `Graph` $\times$ `Shape` are changed so that the parameter of the type `Shape` is deleted. However, one exception is the method `draw` in the class `VerticalRenderer`, for which no change happened. The

```
-- Python 2.7.6                              -- Python 2.7.7

if (co->co_flags & CO_GENERATOR) {           if (co->co_flags & CO_GENERATOR) {
   ...                                          ...
   Py_XDECREF(f->f_back);                       Py_CLEAR(f->f_back);
   f->f_back = NULL;

   PCALL(PCALL_GENERATOR);                      PCALL(PCALL_GENERATOR);

   ...                                          ...
   return PyGen_New(f);                         return PyGen_New(f);
}                                            }
```

Figure 5.4: Code snippets from `ceval.c` in Python 2.7.6 and 2.7.7, respectively.

second rule says that all the methods named `height` in all the classes whose name ends with the string "`Axis`" are renamed to `getHeight`.

While LSdiff uses rules to deduce high-level changes, some conceptual changes are quite tricky and can't be discovered easily. Figure 5.4 presents such an example, which is about changes in `ceval.c` between Python 2.7.6 and 2.7.7. Since `Py_CLEAR` also assigns its argument to `NULL`, there is only one conceptual edit here. However, without the knowledge about `Py_CLEAR`, one would expect that there are two conceptual changes: one changes the function name and the other one removes the assignment.

In summary, we have used three examples to demonstrate that conceptual edits are quite common. What we would like to know is how many conceptual edits typically occur between two consecutive versions for each changed file. It's easy to count the number of primitive edits in each changed file by textual diff, but it's difficult to measure the number of conceptual edits. If we knew the ratio of primitive edits to conceptual edits in general, we could roughly estimate the number of conceptual edits. For that, we need the ratio of primitive edits to conceptual edits and the number of primitive edits in each file changed.

To answer these questions, our first attempt was to reuse the data from an empirical study performed by Kim et a. [Kim et al., 2013]. They investigated the results of using LSdiff to deduce conceptual edits for three projects: carol, dnsjava, and the LSdiff itself. They found that in carol for each pair of consecutive versions, LSdiff deduced 10 rules and 20 facts on average. Here each fact represents a change that is not subsumed in any rule by LSdiff. Thus, if we treat each rule and each fact as a conceptual edit, each pair of consecutive versions contains about 30 conceptual changes. They also reported that each pair of consecutive versions contain about 1229 diff in LOC (Line Of Code), which means that in carol the ratio of primitive edits to conceptual edits is about 40. This ratio for dnsjava and LSdiff itself is 27 and 20, respectively. In carol, 13 files are changed on average in each version. This implies that each changed file contains about $30/13 = 2.31$ conceptual edits. That average number of conceptual edits in each changed file for dnsjava and LSdiff itself is also below 3.

These results indirectly demonstrate the feasibility of PEG, at least for representing changes when we can identify conceptual edits and the source code has become quite stable. However, these results are valid only when we can in fact treat each rule as a single conceptual edit. Also, their results didn't show the worst case about the number of rules and facts for each file in the three projects. For example, it's likely that some specific files can contain quite many conceptual edits. These issues challenge the conclusion about the scalability of PEGs. To address this problem, we have performed an empirical study presented in the next section.

5.2 Empirical study

The study I performed focuses on the Python interpreter. Specifically, I compared each pair of consecutive versions from Python 2.7.1 to 2.7.9, from 3.1 to 3.1.5, and from 3.2 to 3.2.1. Overall, there are 13 pairs. Each version of the Python interpreter contains about 75 files of C source code. For each two consecutive versions, I first wrote a script to filter out all the files that contain changes and also log these changes. For each changed file, I looked through all primitive changes that occurred and identified conceptual changes.

I grouped primitive edits into conceptual edits according to a set of rules. The rules used most frequently are explained below. In other cases, the rules used are minor variations of the presented rules.

1. The primitive edits are considered as one conceptual edit if many operations in the old version are merged into one operation in the new version. This is quite common for API changes. The following code snippet lists such an example from `ceval.c`. The function `Py_CLEAR` includes the functionalities of `Py_XDECREF` and also resets the argument pointer.

   ```
   -- Python 2.7.6               -- Python 2.7.7
   Py_XDECREF(f->f_back);        Py_CLEAR(f->f_back);
   f->f_back = NULL;
   ```

2. The primitive edits are considered as one conceptual edit if a block comment spanning multiple lines is added or removed.

3. The primitive edits are considered as one conceptual edit if changes are about the same macro. The following code snippets list such an example from `dtoa.c`. Note that the extra lines in Python 2.7.7 below were absent in Python 2.7.6. I view this

as one conceptual edit since the lines added are for dealing with the change of the macro `MAX_ABS_EXP`.

```
-- Python 2.7.6                     -- Python 2.7.7
#define MAX_ABS_EXP 19999U          #define MAX_ABS_EXP 1100000000U
                                    #if MAX_ABS_EXP > INT_MAX
                                    #error "MAX_ABS_EXP should fit in an int"
                                    #endif
```

4. The primitive edits are considered as one conceptual edit if multiple changes are due to a variable renaming. For example, the following code lists such an example from `dtoa.c`. These changes are identified as one conceptual edit since they are all about renaming the variable `nd` to `ndigits`.

```
-- Python 2.7.6             -- Python 2.7.7
nd0 = nd = s - s1;          ndigits = s - s1;
...                         ...
nd += s - s1;               ndigits += s - s1;
...                         ...
if (!nd && !lz) ...         if (!ndigits && !lz) ...
```

5. The primitive edits are considered as one conceptual edit if a single `if` statement is added. One example from `sysmodule.c` is given below. I treat these edits as a single conceptual edit because if they are not changed at the same time, the change will cause compiling error.

```
   -- Python 2.7.6          -- Python 2.7.7
                            if (PyErr_Occurred()) {
                                Py_DECREF(version);
                                return NULL;
                            }
```

6. The primitive edits are considered as one conceptual edit if the changes are caused by the change of the signature of some function. The following lists such an example from `ast.c`.

```
-- Python 2.7.7
static PyObject *
parsestr(struct compiling *c, const char *s){
...
if ((v = parsestr(c, STR(CHILD(n, 0)))) != NULL) {
...
s = parsestr(c, STR(CHILD(n, i)));

-- Python 2.7.8
static PyObject *
parsestr(struct compiling *c, const node *n, const char *s){
...
if ((v = parsestr(c, n, STR(CHILD(n, 0)))) != NULL) {
s = parsestr(c, n, STR(CHILD(n, i)));
```

7. The primitive edits are considered as one conceptual edit if the same statement is added at multiple places. They are treated as a single conceptual edit because they have the same functionality. The following example from `ceval.c`

```
-- Python 2.7.7          -- Python 2.7.8
                         if (_Py_Finalizing && tstate != _Py_Finalizing) {
                             PyThread_release_lock(interpreter_lock);
                             PyThread_exit_thread();
                             assert(0);  /* unreachable */
                         }
                         ...
                         if (_Py_Finalizing && _Py_Finalizing != tstate) {
                             PyThread_release_lock(interpreter_lock);
                             PyThread_exit_thread();
                         }
```

8. The primitive edits are considered as one conceptual edit if the changes are about introducing a variable and the corresponding operations on the variable. The following is such an example from `pythonrun.c`. I view these changes as one conceptual change because all the changes are related to the variable `_Py_Finalizing`.

```
-- Python 2.7.7          -- Python 2.7.8
                         PyThreadState *_Py_Finalizing = NULL; -- Global
                         ...
                             _Py_Finalizing = NULL; -- Inside a function
                             ...
                             /* Remaining threads...
                             _Py_Finalizing = tstate;
```

9. The primitive edits are considered as one conceptual edit if a function is added. The whole function is treated as a conceptual edit. One such example from `marshal.c` is shown below.

```
-- Python 2.7.5          -- Python 2.7.6
                         static void
                         w_pstring(const char *s, Py_ssize_t n, WFILE *p){
                                 W_SIZE(n, p);
                                 w_string(s, n, p);
                         }
```

10. The primitive edits are considered as one conceptual edit if the visibility of a sequence of statements is restricted through a use of macros. The following code snippets present such an example from `bltinmodule.c`.

```
-- Python 2.7.3                       -- Python 2.7.4
                                      #ifdef Py_USING_UNICODE
...                                   ...
if (unicode_newline == NULL) {        if (unicode_newline == NULL) {
    Py_CLEAR(str_newline);                Py_CLEAR(str_newline);
    Py_CLEAR(str_space);                  Py_CLEAR(str_space);
    return NULL;                          return NULL;
}                                     }
if (unicode_space == NULL) {          if (unicode_space == NULL) {
    Py_CLEAR(str_newline);                Py_CLEAR(str_newline);
    Py_CLEAR(str_space);                  Py_CLEAR(str_space);
    Py_CLEAR(unicode_space);              Py_CLEAR(unicode_space);
    return NULL;                          return NULL;
}                                     }
                                      #endif
```

11. The primitive edits are considered as one conceptual edit if multiple calls of the same function are replaced by calling another function. For example, the following is such an example from `compile.c`. We observe that all places calling `PyObject_REPR(x)` have been replaced to call the function `PyString_AS_STRING(PyObject_Repr(x))`. Thus they are treated as one conceptual edit.

```
-- Python 2.7.8
PyObject_REPR(c->u->u_ste->ste_id),
PyObject_REPR(c->u->u_ste->ste_symbols),
PyObject_REPR(c->u->u_varnames),
PyObject_REPR(c->u->u_names)
PyObject_REPR(name),
PyObject_REPR(co->co_freevars));

-- Python 2.7.9
PyString_AS_STRING(PyObject_Repr(c->u->u_ste->ste_id)),
PyString_AS_STRING(PyObject_Repr(c->u->u_ste->ste_symbols)),
PyString_AS_STRING(PyObject_Repr(c->u->u_varnames)),
PyString_AS_STRING(PyObject_Repr(c->u->u_names))
PyString_AS_STRING(PyObject_Repr(name)),
PyString_AS_STRING(PyObject_Repr(co->co_freevars)));
```

12. The primitive edits are considered as one conceptual edit if some code of a complicated function has been extracted out to form another function. One example from `sysmodule.c` is given below. The function `sys_getsizeof` in Python 2.7.8 contains 59 LOC. In Python 2.7.9, this function contains about 20 LOC. This is realized by extracting most computations into a new function `_PySys_GetSizeOf`.

```
-- Python 2.7.8
sys_getsizeof(PyObject *self, PyObject *args, PyObject *kwds){
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "O|O:getsizeof",
                                     kwlist, &o, &dflt))
        return NULL;
    ...
    if (PyType_Ready(Py_TYPE(o)) < 0)
        return NULL;
    ...
}

-- Python 2.7.9
size_t _PySys_GetSizeOf(PyObject *o) {
    ...
    if (PyType_Ready(Py_TYPE(o)) < 0)
        return (size_t)-1;
    ...
}
sys_getsizeof(PyObject *self, PyObject *args, PyObject *kwds){
    ...
    if (!PyArg_ParseTupleAndKeywords(args, kwds, "O|O:getsizeof",
                                     kwlist, &o, &dflt))
        return NULL;
    size = _PySys_GetSizeOf(o);
    ...
}
```

13. The primitive edits are considered as one conceptual edit if the definition of a type has changed and other changes in response to that change. The example from `dtoa.c`, presented in Figure 5.5, illustrates this rule. I view these changes as one conceptual edit since the they are all related to the removal of fields of `BCinfo`.

14. The primitive edits are considered as one conceptual edit if multiple statements that are introduced at the same time are removed simultaneously. For example, in

```
-- Python 3.1.1                                      -- Python 3.1.2
struct BCinfo {                                      struct BCinfo {
    int dp0, dp1, dplen, dsign, e0, inexact;
    int nd, nd0, rounding, scale, uflchk;                int e0, nd, nd0, scale;
};                                                   };
...
    dsign = bc->dsign;
    bc.dp0 = bc.dp1 = s - s0;
    bc.dp1 = s - s0;
    bc.dplen = bc.dp1 - bc.dp0;
```

Figure 5.5: Changes related to a type definition are viewed as a single conceptual edit.

```
-- Python 2.7.8                                            -- Python 2.7.9
if (_Py_Finalizing && tstate != _Py_Finalizing) {
    PyThread_release_lock(interpreter_lock);
    PyThread_exit_thread();
    assert(0);  /* unreachable */
}
...
if (_Py_Finalizing && _Py_Finalizing != tstate) {
    PyThread_release_lock(interpreter_lock);
    PyThread_exit_thread();
}
```

Figure 5.6: The edits of removing both statements that were introduced in the previous change are classified as one conceptual change.

Figure 5.6, both `if` statements were introduced in Python 2.7.8 but were removed in Python 2.7.9. Thus they are considered as one conceptual edit.

Based on these rules, Figure 5.7 presents the study results in detail. In the figure, we list the old and new version compared, the number of files changed (*nf*), the total number of primitive changes (*tp*), the total number of conceptual changes (*tc*), the maximum number of primitive changes among all the files changed (*mp*), the maximum number of conceptual edits (*mc*), the ratio of primitive edits to conceptual edits, and the number of conceptual changes per file (*cpf*). Note that we don't present the ratio of *mp* over *mc* since they may not occur in the same file. For example, in the change from 2.7.3

old version	new version	*nf*	*tp*	*tc*	*mp*	*mc*	ratio	*cpf*
2.7	2.7.1	6	42	13	27	4	3.2	2.2
2.7.1	2.7.2	9	359	21	214	12	17.1	2.3
2.7.2	2.7.3	12	106	25	21	5	4.2	2.1
2.7.3	2.7.4	16	546	51	260	10	10.7	3.2
2.7.4	2.7.5	1	5	1	5	1	5.0	1
2.7.5	2.7.6	5	84	6	51	2	14	1.2
2.7.6	2.7.7	4	59	6	49	3	10.0	1.5
2.7.7	2.7.8	6	49	7	17	2	7.0	1.2
2.7.8	2.7.9	9	218	17	80	4	12.9	1.9
3.1	3.1.1	5	37	13	13	6	2.8	2.6
3.1.1	3.1.2	16	1445	117	769	43	12.4	7.3
3.1.3	3.1.4	8	122	13	63	6	9.4	1.6
3.1.4	3.1.5	2	12	7	8	4	1.7	3.5
3.2	3.2.1	11	258	36	67	10	7.2	3.3
overall		110	3342	333	769	43	10.0	3.0

Figure 5.7: Results of the empirical study of identifying conceptual edits. The number *nf* denotes the number of files changes. The columns *tp* and *tc* denote the total number of primitive edits and conceptual edits, respectively. The ratio in the table is computed as tp/tc. The columns *mp* and *mc* denote the number of maximum primitive and conceptual edits, respectively. The column *cpf* shows the number of conceptual changes per file.

to 2.7.4, two files contain 10 conceptual changes whereas their primitive changes differ significantly. While one contains 260 primitive changes, the other contains only 42.

From the figure, we observe that the average ratio of primitive edits to conceptual edits is 10. Also, the average number of conceptual edits is only 3. While most files contain very few conceptual edits, the file `dtoa.c` has 43 conceptual changes from version 3.1.1 to 3.1.2. The number of primitive edits for that file is 769.

To give a better idea about the number of conceptual edits in files, Figure 5.8 shows the percentage of all files against number of conceptual edits. Each point (x,y) in the

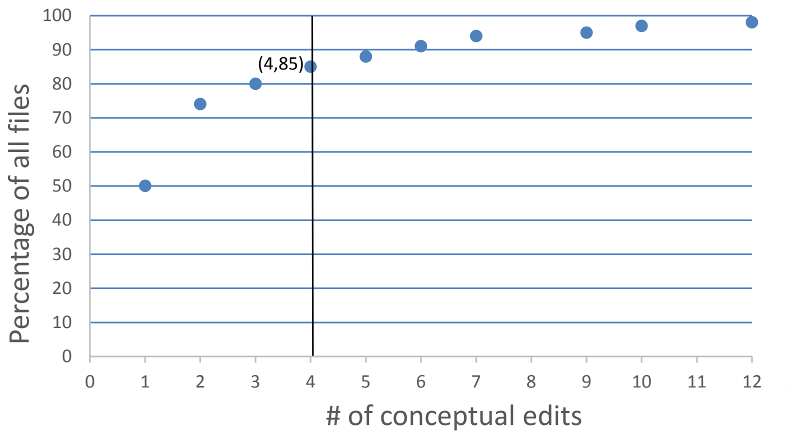


Figure 5.8: Number of conceptual changes over all the files. Note that two points (31,99) and (43,100) have been omitted in the figure to make the rest of the figure more readable.

graph denotes that y% of all files have fewer than x conceptual edits. For example, the point (4,85) represents the fact that 85% of all the files contain at most 4 conceptual edits. A PEG can handle 4 number of conceptual edits very well because it contains at most 16 nodes.

Figures 5.7 and 5.8 show that while some files may contain quite many conceptual edits, the frequency of that happening is very low. For example, in the 110 files I investigated, only 2 files contain more than 12 edits. Also, 94% of all the files contain fewer than 7 conceptual edits. Thus, we conclude that PEGs are scalable in practice, at least when source code is relatively stable. In cases with more than 4 independent edits, we can employ the concept of branch filters to reduce the complexity of PEGs.

Chapter 6: Conclusions

In this project, I have proposed a choice edit model as a new foundation for reasoning about edits and supporting more flexible selective undo, in particular, partial selective undo, which wouldn't be possible without untangling entangled edits. Based on the choice calculus, the edits in the choice edit model are highly compositional, allowing edits to be transformed systematically and identified for undoing without affecting other edits. Another advantage of the choice edit model is that discovering relations among edits is made simpler and formal. Note that the inability of precisely detecting dependencies among edits is a main obstacle for supporting partial selective undo.

This project has further presented the concept of PEGs to facilitate users of reaping the full benefits of the choice edit model. In particular, they make the hidden relations between program edit, undo, and redo operations, and the resulting programs explicit. PEGs are dense and this fact reflects that the choice edit model is very flexible in supporting selective undo and redo.

A potential challenge with PEGs is their high complexity. Concretely, a PEG can have an exponential number of nodes with respect to the number of dimensions in the expression. I have argued that this is often not a big problem in practice if we represent conceptual changes. Through an analysis of the Python interpreter source code, our empirical study showed that above 85% of all changed files contain fewer than 4 conceptual changes, demonstrating the feasibility of PEGs. Moreover, we can handle complexity by making PEGs modular through branch filters. Overall, we view the choice edit model and PEGs as viable foundations for the study of powerful undo operations.

Bibliography

J. E. Archer, Jr., R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Trans. Program. Lang. Syst.*, 6(1):1–19, Jan. 1984.

T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1(3):269–294, 1994.

A. G. Cass and C. S. Fernandes. Using task models for cascading selective undo. In *Task Models and Diagrams for Users Interface Design*, LNCS 4385, pages 186–201. 2007.

S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM Int. Conf. on Functional Programming*, pages 29–40, 2012.

S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *ACM Trans. on Programming Languages and Systems*, 36(1):1:1–1:54, 2014.

J. Dagit and M. J. Sottile. Identifying change patterns in software history. *CoRR*, abs/1307.1719, 2013.

S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, pages 166–177, 2000.

D. Dig, K. Manzoor, R. Johnson, and T. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 427–436, May 2007.

W. K. Edwards, T. Igarashi, A. LaMarca, and E. D. Mynatt. A temporal model for multi-level undo and redo. In *ACM Symp. on User Interface Software and Technology*, pages 31–40, 2000.

M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering*, LNCS 7680, pages 55–100, 2013.

M. Erwig, K. Smeltzer, and K. Xu. Putting Edit Histories to Work: Discovering Program Families in Programs. 2015. Draft paper.

C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. pages 773–792, 2012.

M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 309–319, 2009.

M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *Software Engineering, IEEE Transactions on*, 39 (1):45–62, Jan 2013.

G. B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1):50–87, 1986.

B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 260–267, 1996.

S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, 2012a.

S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *European Conf. on Object-Oriented Programming*, pages 79–103, 2012b.

S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 552–576, 2013.

S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining fine-grained code changes to detect unknown change patterns. pages 803–813, 2014.

H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 302–321, 2010.

A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, Dec. 1994. ISSN 1073-0516.

M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '99, pages 131–139, 1999. ISBN 1-58113-065-1.

B. Shao, D. Li, and N. Gu. An algorithm for selective undo of any operation in collaborative applications. In *ACM Int. Conf. on Supporting Group Work*, pages 131–140, 2010.

C. Sun. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Hum. Interact.*, 9(4):309–361, Dec. 2002. ISSN 1073-0516.

D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(10):1454–1470, Oct. 2009.

J. Vitter. US&R: A new framework for redoing. *IEEE Software*, 1(4):39–52, Oct 1984.

E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, GPCE 2014, pages 29–38, 2014.

S. Weiss, P. Urso, and P. Molli. An undo framework for p2p collaborative editing. In *Collaborative Computing: Networking, Applications and Worksharing*, volume 10 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 529–544. 2009.

Y. Yang. Undo support models. *Int. Journal of Man-Machine Studies*, 28(5):457–481, 1988.

A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. on Software Engineering*, 30(9):574–586, 2004.

Y. Yoon and B. A. Myers. Supporting selective undo in a code editor. In *International Conference on Software Engineering (ICSE'15)*. To appear.

Y. Yoon and B. A. Myers. Capturing and analyzing low-level events from the code editor. In *ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 25–30, 2011.

Y. Yoon and B. A. Myers. An exploratory study of backtracking strategies used by developers. In *Workshop on Cooperative and Human Aspects of Softw. Eng.*, pages 138–144, 2012.

Y. Yoon and B. A. Myers. A longitudinal study of programmers backtracking. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 101–108, 2014.

Y. Yoon, B. Myers, and S. Koo. Visualization of fine-grained code change history. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 119–126, 2013.

C. Zhou and A. Imamiya. Object-based nonlinear undo model. In *Int. Conf. on Computer Software and Applications*, pages 50–55, 1997.