AN ABSTRACT OF THE DISSERTATION OF

Eric Walkingshaw for the degree of Doctor of Philosophy in Computer Science presented on June 13, 2013.

Title: The Choice Calculus: A Formal Language of Variation

Abstract approved: ____________________
Martin Erwig

In this thesis I present the choice calculus, a formal language for representing variation in software and other structured artifacts. The choice calculus is intended to support variation research in a way similar to the lambda calculus in programming language research. Specifically, it provides a simple formal basis for presenting, proving, and communicating theoretical results. It can serve as a common language of discourse for researchers working on different views of similar problems and provide a shared back end in tools.

This thesis collects a large amount of work on the choice calculus. It defines the syntax and denotational semantics of the language along with modular language extensions that add features important to variation research. It presents several theoretical results related to the choice calculus, such as an equivalence relation that supports semantics-preserving transformations of choice calculus expressions, and a type system for ensuring that an expression is well formed. It also presents a Haskell DSEL based on the choice calculus for exploring the concept of variational programming.

The Choice Calculus: A Formal Language of Variation

by

Eric Walkingshaw

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 13, 2013
Commencement June 2014

Doctor of Philosophy dissertation of Eric Walkingshaw
presented on June 13, 2013.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

Eric Walkingshaw, Author

ACKNOWLEDGEMENTS

First and foremost, thank you to my wife Allison for her love, patience, flexibility, and support over the last six years. Thank you for moving away from home with me, for taking care of me when I'm busy, for inspring me every day with your own success and hard work, and most of all, for sharing your life with me.

A very special thank you also to my advisor Martin Erwig, who has contributed tremendously to my development as a researcher and thinker. I am grateful for your confidence in me, consistent enthusiasm, guidance, and unwavering support. Thank you for securing funding for me and for everything else you have done on my behalf. I look forward to many future collaborations.

A heartfelt thank you to Ronnie Macdonald and Laurie Meigs for their kindness and financial support through the ARCS Foundation. Your generosity has meant so much to both Allison and me, and I truly loved getting to know you both and your families. Ronnie passed away last year—she was witty and warmhearted and she had a profound impact on my life in the time that I knew her.

Thank you to the many faculty members at Oregon State who have influenced me during my time here. Thank you to the members of my program committee—Margaret Burnett, Alex Groce, Ron Metoyer, Maggie Niess, and Chris Scaffidi—for all of their support and advice. I look forward to working with you for years to come! Thank you also to Paul Cull for his humor and candor over coffee every Friday.

Thank you to Christian Kästner and Shriram Krishnamurthi for comments on draft papers, advice, and interesting discussions at Dagstuhl and elsewhere. Thank you to Sven Apel and Alexander von Rhein for hosting my visit to Passau last summer. Thank you to Tillmann Rendel and Klaus

Ostermann for collaborating with us on the selection extension, and to Klaus for hosting me next year in Marburg.

Thank you to the many friends we have made in Corvallis. We arrived knowing nobody and now have so many wonderful friendships. I especially want to thank Chris, Chris, Andrew, Katie, and Will for being the best group of friends we could ever have wished for. I will miss our game nights, beer brewing, Resistance arguments, and Friend Thanksgivings immensely. I also want to thank Adaline and Alex for amazing rafting trips; George, Dick, and Mike for Wednesday night gaming; Sheng for so many interesting debates; Rob and Tim for so many shared beers; Todd for his infectious kindness; and the whole KEC Krew basketball team. I'm sorry for not listing everyone here—it would be impossible. Thank you all for your friendship.

Thank you to my family for their unconditional love and support throughout my life, especially Mom, Dad, Heidi, Grandma, and Brent. Thank you to Jill and Lindsay for being such incredibly generous and loving in-laws. I would especially like to thank my mom for cultivating in me a lifelong love of learning, without which this thesis would not exist.

Finally, thank to the funding adjacencies who supported this work, namely, the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and the National Science Foundation under the grants CCF-0917092 and CCF-1219165.

CONTRIBUTION OF AUTHORS

This thesis draws material from several papers that were co-authored with Martin Erwig [Erwig and Walkingshaw, 2010, 2011a,b, 2012a,b, Walkingshaw and Erwig, 2012]. Chapter 5 is based on a collaboration with Tillmann Rendel and Klaus Ostermann [Erwig et al., 2013a].

I have also worked closely with Sheng Chen on applying the choice calculus to variational type inference [Chen et al., 2012, 2013], and with Duc Le on applying the choice calculus to improving program understandability [Le et al., 2011], although these works are described only briefly in Chapter 9.

I also must acknowledge Martin as the source of the dessert joke in Section 7.2.3, though I wish I'd thought of it.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Motivation and Impact 2

1.2 Contributions and Outline of this Thesis 6

2 Background 11

2.1 Object Language vs. Metalanguage 11

2.2 Modeling, Implementation, and Configuration 13

2.3 Approaches to Representing Variation 15

3 The Choice Calculus 23

3.1 Introduction to the Choice Calculus 24

3.2 Design of a Variation Representation 26

3.2.1 Semantics-Driven Design 26

3.2.2 Synchronizing Variation Points 28

3.2.3 Alternative vs. Optional Variation 30

3.3 Syntax of the Choice Calculus 32

3.3.1 Representing the Object Language 32

3.3.2 Structuring the Variation Space 35

3.3.3 Well Formedness and Other Static Properties 38

3.4 Compositional Denotational Semantics 40

3.5 Semantics-Preserving Transformations 44

3.5.1 Equivalence Rules 46

3.5.2 Instantiating the Equivalence Rules 50

3.5.3 Dimension and Choice Normal Forms 52

3.6 Variation Design Theory 54

3.6.1 Syntactic vs. Semantic Design Criteria 55

3.6.2 Equivalent Alternatives and Tags 58

3.6.3 Removing Pseudo-Choices and Pseudo-Dimensions . . 62

3.6.4 Removing Dominated Choices 64

4 Extensions to Support Reuse 66

4.1 Motivation 67

4.1.1 Reuse of Variation Points 68

4.1.2 Reuse of Alternatives 69

TABLE OF CONTENTS (Continued)

Page

4.1.3 Reuse of Common Subexpressions 71
4.1.4 Reuse of Variational Components 74
4.2 Staging Relative to Dimension Elimination 76
4.3 Syntax and Well-Formedness 78
4.4 Denotational Semantics 80
4.5 Semantics-Preserving Transformations 83
4.5.1 Transformations Involving Sharing 84
4.5.2 Transformations Involving Macros 88
4.5.3 Commuting the Two Extensions 90

5 Internalized Selection 92
5.1 Representing Selection in the Choice Calculus 93
5.2 Motivation 94
5.2.1 Dimension and Tag Renaming 95
5.2.2 Modularity of Variational Components 97
5.2.3 Operational Semantics 98
5.3 Design Questions 99
5.3.1 Undeclared Dimensions 99
5.3.2 Multiple Matching Dimensions 100
5.3.3 Undeclared Tags 104
5.3.4 Dependent Dimensions 105
5.3.5 Staging and Scope of Selection 109
5.4 Syntax and Denotational Semantics 112
5.5 Configuration Type System 114
5.5.1 Configuration Types 115
5.5.2 Typing Environment and Important Properties 116
5.5.3 Typing Rules 118
5.6 Semantics-Preserving Transformations 120

6 Compositional Choice Calculus 124
6.1 Motivation 125
6.1.1 Compositional Approaches 126
6.1.2 Annotative Approaches and the Choice Calculus . . . 128
6.1.3 Representing Feature Interactions 131

TABLE OF CONTENTS (Continued)

Page

6.2 Integrating the Two Approaches 133

6.3 Adding Variation Abstractions 136

6.3.1 Reusable Optional Wrappers 136

6.3.2 Variation Metaprogramming 138

6.4 The Compositional Choice Calculus 139

6.5 Denotational Semantics 143

6.5.1 Challenges for a Formal Semantics 143

6.5.2 Dimension and Choice Elimination 146

6.5.3 Composing Partial Semantics 147

6.5.4 Computing the Semantics 149

6.6 Comparison of Relative Local Expressiveness 150

7 Variational Programming 155

7.1 A Variation DSEL in Haskell 157

7.2 Variational Lists 165

7.2.1 Representing Variational Lists 167

7.2.2 Standard Variational List Functions 173

7.2.3 Edit Operations for Variational Lists 179

7.2.4 Variational Programming Modes 187

7.3 Variational Software 188

7.3.1 Representing Variational Haskell 189

7.3.2 Edit Operations for Variational Haskell 193

8 Related Work 200

8.1 Comparison with the C Preprocessor 201

8.2 Organizing the Variation Space 205

8.3 Internalized Selection 209

8.4 Combining Annotative and Compositional Variation 210

9 Conclusion 213

9.1 The Choice Calculus in Practice 213

9.1.1 Application to Variational Typing 214

9.1.2 Other Applications 218

9.2 Summary of Contributions 220

TABLE OF CONTENTS (Continued)

Page

9.3 Future Work 221

Appendix A Proof of Equivalence Laws 224

A.1 Syntax Definition 224

A.2 Semantics Definition 225

A.3 Proofs that the Laws Preserve the Semantics 226

LIST OF FIGURES

Figure Page

2.1 Summary of trade-offs between representation approaches. . . 17

3.1 Core choice calculus syntax. 33
3.2 Variational lambda calculus. 34
3.3 Variational lists. 35
3.4 Choice calculus with local dimension declarations. 36
3.5 Well-formedness judgment. 38
3.6 Bound and free dimensions. 40
3.7 Denotational semantics of some example expressions. 41
3.8 Definition of choice elimination. 42
3.9 Denotational semantics of choice calculus expressions. 43
3.10 Three semantically equivalent expressions. 45
3.11 Choice commutation rules. 46
3.12 Choice simplification rules. 47
3.13 Dimension-structure commutation. 49
3.14 Properties of the equivalence relation. 49
3.15 Chc-Obj equivalence rule instantiated by lambda calculus. . . 50
3.16 Dim-Obj equivalence rule instantiated by lambda calculus. . . 51
3.17 Proof that $C\langle B\langle 1,3\rangle, B\langle 2,4\rangle\rangle \equiv B\langle C\langle 1,2\rangle, C\langle 3,4\rangle\rangle$. 56
3.18 Semantics of expression *abc*. 57

4.1 Alternative reuse in CPP. 70
4.2 Optional wrapper in CPP. 73
4.3 Reuse of a variational component using **macro**. 75

LIST OF FIGURES (Continued)

Figure Page

4.4 Varying the parameter names of `twice` and `thrice`. 77

4.5 Syntax of share and macro extensions. 79

4.6 Extension of bound and free dimensions. 79

4.7 Free variables. 80

4.8 Extended well-formedness judgment. 81

4.9 Extension of choice elimination. 81

4.10 Denotational semantics of share and macro extensions. 84

4.11 Sharing introduction/elimination rules. 85

4.12 Choice-sharing commutation rules. 85

4.13 Dimension-sharing commutation rules. 86

4.14 Remaining sharing commutation rules. 87

4.15 Macro introduction/elimination rules. 88

4.16 Commuting macros with choices and dimensions. 89

4.17 Remaining macro commutation rules. 90

4.18 Commuting sharing and macros. 91

5.1 Syntax of selection extension. 112

5.2 Denotational semantics of selection extension. 113

5.3 Configuration types. 115

5.4 Configuration type concatenation. 116

5.5 Configuration typing environment. 117

5.6 Configuration typing rules. 118

5.7 Configuration type reduction. 120

LIST OF FIGURES (Continued)

Figure | Page

5.8 Selection introduction/elimination and commutation. 121

5.9 Choice and dimension commutation rules. 122

5.10 Commuting selection and object structures. 123

6.1 A small integer buffer SPL. 127

6.2 Class refinements implementing undo features. 128

6.3 Buffer with annotated logging and undo features. 129

6.4 Modularized feature interactions. 132

6.5 Undo refinement u with annotative variation. 134

6.6 Syntax of the compositional choice calculus. 140

6.7 New equivalence laws for CCC. 141

6.8 Extended choice elimination. 146

6.9 Partial semantics values. 147

6.10 Partial semantics composition. 148

6.11 Computing partial semantics. 149

6.12 Relative local expressiveness of choice calculus variants. . . . 154

7.1 The choice calculus as a Haskell data type. 158

7.2 Monad instance for the `V` type constructor. 164

7.3 Smart constructors for expanded and variational lists. 170

7.4 Menu preferences as a variational list. 171

7.5 Alternative definition of `menu` with tagged alternatives. 173

7.6 Computing the variational length of an expanded list. 175

7.7 Concatenating expanded lists. 177

LIST OF FIGURES (Continued)

Figure		Page
7.8	Indexing into an expanded list.	178
7.9	Right fold on an expanded list.	179
7.10	Dimension hoisting operation (unsafe).	183
7.11	Capture-avoiding dimension hoisting operation.	184
7.12	Transformation to eliminate a class of redundant decisions.	185
7.13	Definition of variational Haskell.	190
7.14	Auxiliary definitions for variational Haskell programs	191
8.1	Correspondence of choice calculus and CPP concepts.	203
8.2	Ambiguity of internal configuration in CPP.	205
9.1	Variational types.	215
9.2	Typing rule for applications in VLC.	216
9.3	Graphical representation of a choice calculus expression.	219

Dedicated to the memories of
Ronnie Macdonald and Gene Walkingshaw

Chapter 1 – Introduction

A core tenet of software engineering is the benefit of *software reuse* [Krueger, 1992]. Current research on software product lines [Pohl et al., 2005], generative programming [Czarnecki and Eisenecker, 2000], and feature-oriented software development [Apel and Kästner, 2009] seek to maximize reuse within a family of related programs by promoting and enabling the development of *massively variational software*. A massively variational software system represents a potentially huge number of related program variants. Each variant can be individually generated from a shared set of resources to include a particular set of features or run in a particular environment. In this way, variational software can be used to efficiently produce and maintain programs that are customized for different platforms, domains, tasks, or even individual users.

The major contribution of this thesis is the *choice calculus*, a fundamental representation of variation that supports foundational research on the creation, evolution, analysis, and verification of massively variational software. The choice calculus is a formal language that is *simple*, *generic*, *extensible*, and *instantiable* in order to support theoretical research on a wide range of applications [Erwig and Walkingshaw, 2010, 2011b]. In the next section we define and motivate each of these qualities. We also argue why the choice calculus is needed and describe the potential long-term impact of this work on the state of software variation research.

In addition to a formal description of the choice calculus itself, this thesis also presents several language extensions and theoretical results related to the choice calculus. Section 1.2 enumerates the specific contributions in the context of an outline of the structure of the rest of the thesis.

1.1 Motivation and Impact

The development of massively variational software is an active area of research with a huge variety of goals and challenges. Essentially any traditional software engineering or programming language issue is complicated by the presence of variation and so has its own line of research in the software product line community. For example, in addition to creating variational software, there is work on evolving [Figueiredo et al., 2008], testing [Cohen et al., 2008, Perrouin et al., 2010], parsing [Kenner et al., 2010], model checking [Classen et al., 2011], and typing [Apel et al., 2010, Kästner et al., 2012a] variational software. These examples are of course just the tip of the iceberg.

Aside from software product lines and related research, effectively dealing with variation is important in many other areas of computer science. For example, version control systems are concerned with managing variation in software over time [Tichy, 1985], metaprogramming systems provide a way to programmatically vary software at compile-time [Sheard, 2001], and union types are used to indicate that the runtime representation of a value can vary [Pierce, 1991]. There is also a huge range of applications for variational data structures. Variational trees can form the basis for search algorithms, while variational graphs are useful for navigation and path-finding systems.

The amount and diversity of this work has naturally led to the development of an equally diverse set of tools and languages for supporting it. Each tool or language reflects a particular view of the shared problem of managing variation, each with its own way of indicating which parts of an artifact vary and how a particular variant is produced. While the diversity of variation representations is not inherently a problem, there are many opportunities for the reuse of ideas and theoretical results between these fields that are missed by current approaches. Indeed, it is not clear whether researchers in many of these disparate areas even recognize their work as facets of a common problem. Additionally, by focusing on specific instances of the problem and using languages tailored to those instances, researchers may miss insights that can only be gained by consider variation from a broader, more abstract point of view.

The choice calculus addresses these issues somewhat paradoxically by adding "yet another language" to the pile. However, it has several qualities that make it well-suited as a *common language of discourse* and *shared research platform* for researchers working on different views of the problem of effectively dealing with variation.

1. The choice calculus is *generic*. That is, it is a metalanguage for describing variation that is independent of any particular object language (see Section 2.1). This is important for two reasons. First, it allows the choice calculus to be applied in a broad range of contexts and to many different artifact types. Second, by abstracting away from the details of specific programming languages and data formats, it supports the development of transformations and theoretical results that can be reused across applications and fields. Genericity is achieved in the choice calculus by the *object structure* construct described in Section 3.3.1, which encodes the underlying artifact being varied as an abstract tree.

2. On the other hand, the choice calculus is *instantiable*. That is, the generic abstract tree model of the core choice calculus can be replaced by the specific abstract syntax of any object language. This is important since details of the object language are of central importance for some applications, such as many kinds of variational analyses. For example, a variational type inference algorithm must necessarily take into account how to infer types in the underlying object language (see Section 9.1.1). Section 3.3 shows how the choice calculus can be instantiated by an arbitrary object language.

3. The choice calculus is *simple*. That is, it provides a direct and minimalistic view of variation that is easy to analyze and manipulate. The representation is direct in the sense that it encodes variation extension-

ally by enumerating the differences between variants (see Section 2.3). The choice calculus is minimalistic in that the core language provides just three orthogonal constructs—*choices* for locally capturing points of variation; *dimensions* for scoping, organizing, and synchronizing choices; and *object structures* for encoding the object language, as described above. The small number of constructs is important since it minimizes the number of cases that must be considered in rigorous theoretical work. Orthogonality is also a desirable language quality for formal reasoning [Scott, 2009, p. 328]. More specifically, orthogonality is important for the choice calculus since it supports a compositional semantics that is critical to the next quality.

4. While the choice calculus is simple, it is also *extensible*. That is, new language features can be modularly added to the simple core. This is important since some applications will require variation features not provided by the minimalistic core. The compositional semantics of the choice calculus allows the language to be extended in a way that preserves existing definitions and results. Chapter 4 and Chapter 5 demonstrate how the choice calculus can be extended to provide new language features. These modular extensions can be arbitrarily included or not in a particular application of the choice calculus. More importantly, however, they provide a template for other researchers to define their own extensions. The simplicity of the choice calculus combined with its extensibility enable researchers to consider only the precise set of features that they need, minimizing the number of cases that must be considered in rigorous theoretical work.

The genericity and simplicity of the choice calculus supports *analytical rigor*. By stripping the representation of variation down to its barest form, we can identify and prove basic facts and relationships between variational expressions that can be reused in a broad range of contexts. Many such results are provided in Chapter 3. However, the ability to instantiate the choice calculus with different object languages and extend it with different language features, means that the choice calculus can also be applied in contexts where more specific or featureful languages are needed. In this way, the choice calculus describes a family of related languages for representing, transforming, and analyzing variation. Since these languages are all based on a shared core, ideas and results can be shared more easily between them.

The value of having a language to fill this role in variation research is evidenced by the lambda calculus in programming language and type theory research. In this field, a new typing feature can be presented by systematically extending the lambda calculus with a new construct, then extending a set of standard typing rules to type that construct. This allows other researchers familiar with the lambda calculus to see the effects of the new feature immediately and possibly to combine it with other extensions developed in a similar way. In this way, building on a common theoretical foundation lowers the barrier of entry to working on harder, more interesting problems, and makes sharing and reusing the results of that research easier.

Although informed by existing research, the choice calculus is an attempt to address the variation management problem from first principles. While solving a concrete problem has the benefit of being immediately useful, a general, theoretical perspective can reveal underlying patterns, expose uncovered territory, and suggest new solutions for old problems. A long-term sign of success will be if these insights percolate back into more application-oriented research. In fact, we have observed success of this kind already with the choice calculus. In a recent paper on type-checking `#ifdef`-annotated C programs, Kästner et al. write: "the implementation of our type-checking mechanism inside modules with alternative types was particularly inspired by the structures of the choice calculus" [2012b].

The choice calculus and its associated theory can also directly support the construction of tools by providing an established core to build on. The advantages of building tools around a shared core are similar to the advantages for research. Libraries of reusable code can be developed, lowering the barrier of entry to new tools and ensuring that basic functionality is consistent and correct. The choice calculus can also serve as an intermediate language to support interoperability between tools.

Chapter 9 briefly describes how we have successfully applied the choice calculus to solve the problem of extending the Damas-Milner type-inference algorithm to variational programs [Chen et al., 2012, 2013]. This provides a case study illustrating many of the benefits claimed above. In particular, we used multiple different instantiations of the choice calculus and were able to reuse many of the theoretical results from our previous work [Erwig and Walkingshaw, 2011b].

1.2 Contributions and Outline of this Thesis

The high level goal of this thesis is to present the choice calculus in a way that supports its use by other variation researchers. Therefore, in addition to the formal definition of the language itself and its various extensions, for each language feature we also discuss the rationale behind important design decisions and explore alternative definitions. This not only motivates our particular design, but also suggests avenues for future research. We also present several operations on choice calculus expressions and theoretical results that form an initial base of knowledge that can be reused in future applications. In later chapters, we show how the choice calculus can be applied, demonstrating its value and providing a template for future users of the language.

The rest of this section describes the structure of this thesis, enumerating the specific contributions that each chapter makes.

Chapter 2 (*Background*) introduces several concepts and terms that will be used throughout the thesis. It distinguishes between object languages and metalanguages, describes the major roles of languages in existing variation research, and discusses the major approaches to representing variation. It also also motivates some of the high-level design decisions underlying the choice calculus.

Chapter 3 (*The Choice Calculus*) introduces the choice calculus as a simple formal representation of static, annotative variation in tree-structured data [Erwig and Walkingshaw, 2011b]. In the choice calculus, a *choice* encodes a variation point in the tree while *dimensions* structure and synchronize choices. In addition to the language itself, this chapter makes several important contributions.

1. A syntactic well-formedness property on choice calculus expressions that ensures that every choice is bound by a corresponding dimension and has the correct number of alternatives.

2. A denotational semantics for the choice calculus. The denotation of a choice calculus expression is a mapping from decisions to the individual plain tree variants those decisions produce. The semantics is defined compositionally in order to support modular language extensions.

3. A complete set of semantics-preserving transformation laws for choice calculus expressions, enabling the merger or commutation of syntactic forms within choice calculus expressions. A mechanized proof of these laws for a simplified version of the choice calculus is provided in Appendix A.

4. The identification of strategic normal forms for core choice calculus expressions. These can serve as representatives of the equivalence classes defined by the transformation laws. They also have practically desirable properties, such as maximizing sharing between variants.

5. A semantics-based design theory for identifying and removing "dead" subexpressions and superfluous variation in core choice calculus expressions.

Less formally, the chapter provides a rationale for the design of the choice calculus and a comparison with other annotative variation representations. It also informally describes how the choice calculus (and its associated properties and transformations laws) can be instantiated by different object languages.

Chapter 4 (*Extensions to Support Reuse*) presents two extensions of the choice calculus with language features to support reuse. One extension supports the sharing of subexpressions that have already been configured, while the other supports the reuse of subexpressions that can be configured separately at each point of use. The chapter provides a detailed discussion of the motivation and design challenges associated with these extensions. The syntax and denotational semantics of the extensions are formally defined in a modular way with respect to the core choice calculus, enabling the extensions to be included arbitrarily together or separately without changing the existing language definition. The well-formedness property is also extended to the new features. Finally, several new transformation laws are introduced to commute the new extensions with each other and with the existing syntactic forms of the core choice calculus.

Chapter 5 (*Internalized Selection*) presents an extension of the choice calculus with a language feature to support the configuration of choice calculus expressions from within the choice calculus itself [Erwig et al., 2013a]. We provide a detailed discussion of the design challenges associated with this extension, and motivate the chosen design. As with the reuse extensions, this extension is formally defined in a modular way, so that it can be included or not in the choice calculus without changing the existing language or other modular extensions. We also introduce new transformation laws to commute the new extension with existing syntactic forms. The most significant contribution of this chapter is an extension of the well-formedness property into a *configuration type system* that both ensures that an expression is well formed and reveals its dimension structure by its associated configuration type.

Chapter 6 (*Compositional Choice Calculus*) presents the compositional choice calculus (CCC), a formal language that extends the choice calculus with metaprogramming and compositional variation features [Walkingshaw and Erwig, 2012]. The chapter motivates the design of CCC and illustrates how it can be used to address several recurring problems related to effectively representing variation. Formal contributions include a denotational semantics for CCC, and a formal demonstration that the language unifies the annotative and compositional approaches to variation implementation, addressing an open problem in FOSD [Kästner and Apel, 2009]. This demonstration shows that CCC is more locally expressive [Felleisen, 1991] than either approach in isolation.

Chapter 7 (*Variational Programming*) presents a domain-specific language based on the choice calculus and embedded in Haskell to support *variation programming* [Erwig and Walkingshaw, 2012a]. This language represents an application of the choice calculus in a less formal and more exploratory setting. The chapter introduces the idea of *variational data structures*, which have potentially many applications in a huge range of contexts. The chapter explores the representation and manipulation of variational lists in some detail. Finally, it explores how variational Haskell programs can be created and edited from within the DSL, providing a glimpse of how the choice calculus can support sophisticated tools for working with variation software.

Chapter 8 (*Related Work*) collects research related to each of the above contributions that was not discussed in context or in Chapter 2. It also provides a detailed comparison of the choice calculus to the C Preprocessor, which is the most used annotative variation tool in practice.

Finally, Chapter 9 (*Conclusion*) briefly presents some successful applications of the choice calculus that demonstrate its viability as a platform for research on variational software. In particular, it describes how we were able to reuse many results and theoretical machinery from Chapter 3 in the definition of a type system and type inference algorithm for *variational* *lambda calculus*. The chapter closes with a summary of the most important contributions and some directions for future work.

Chapter 2 – Background

At the core of this thesis is the choice calculus, an annotative metalanguage for describing the implementation and organization of static variation in tree-structured artifacts. The goals of this chapter are twofold: first, to establish a context and terminology for discussing variation languages like the choice calculus, which will make the meaning of the above description clear; second, to motivate the high-level design decisions that led us to choose this as a foundation for a general theory of variation.

Section 2.1 describes the differences between object languages and meta-languages, and Section 2.2 discusses the three main roles of metalanguages in the context of creating and managing variation. Section 2.3 describes three approaches to representing variation in software, discusses desirable qualities for a variation representation, and analyzes the trade-offs between each of the three approaches in terms of these qualities. Section 2.3 also argues in favor of a structured annotative approach as a basis for the choice calculus on the grounds that the qualities it supports are most important for the goal of establishing a rich theory of variation.

2.1 Object Language vs. Metalanguage

In principle, we can make any kind of artifact variational. We can use a type constructor V to represent the incorporation of variability into an otherwise non-variational artifact. Thus, a variational artifact of type $V\ a$ (that is, V applied to a) represents potentially many different plain artifacts of type a. For example, a variational Java program represents many different plain Java programs. The creation of a variational artifact typically involves the interplay of more than one language, and we can distinguish between two kinds of languages: (1) *object languages* used to describe the artifacts themselves (corresponding to the type a), and (2) *metalanguages* used to describe the variability within them (corresponding to V).

1. An object language is a language used to describe a single, non-variational artifact. Typical programming languages like C, Java, or Haskell are object languages. More generally, we can consider the abstract syntax of data structures like trees or graphs to be object languages. A variational graph would then represent many different specific graphs, each of which would be represented in the object language of the abstract syntax of graphs.

2. A metalanguage (also called a *variation language*), is a language used to describe the variability in a variational artifact. There are many possible relationships between an object language and a metalanguage; these will be discussed in the rest of this chapter. One possibility is that the metalanguage can be embedded within the object language. For example, the C Preprocessor language (CPP) [CPP] is a widely used metalanguage for describing variability. Through the use of its conditional compilation directives—`#ifdef`, `#if`, `#elif`, `#else`, and `#endif`—object language code can be conditionally included or not in the preprocessed program, depending on the values of various user- and system-defined variables used in the conditions.

A CPP-annotated Java program is therefore a variational Java program, where CPP is the metalanguage and Java is the object language. Running the preprocessor produces a single, plain Java program that can be compiled and run in the usual way. This program is just one of potentially many different variants that can be generated from the variational program.

This thesis focuses on developing and extending the choice calculus, an annotative metalanguage for describing static variation in a similar way as CPP. In general, the choice calculus can be applied to any object language with a tree-structured syntax. Section 3.3.1 describes the abstract representation of object languages in the choice calculus, which is based on a generic representation of abstract syntax trees.

With compositional metalanguages (see Section 2.3), knowing the syntactic structure of an object language is not sufficient to encode variation. These approaches must also be able to combine different pieces of object language code in order to assemble a particular variant. An example of this is aspect weaving in aspect-oriented programming [Kiczales et al., 1997]. This leads to a tighter coupling and blurrier separation between object and metalanguage; for example, the aspect-oriented metalanguage AspectJ [Kiczales et al., 2001] is tightly coupled to the object language of Java. Chapter 6 describes an extension to the choice calculus to support compositional variation. This includes a parameterized composition operator that allows us to maintain a clear separation between object and metalanguage, and to still treat object languages in a generic way.

2.2 Modeling, Implementation, and Configuration

Variation languages can be further organized according to their role in the process of creating and managing variational artifacts. Many metalanguages fulfill multiple roles and so cannot be uniquely classified, but it is instructive to consider each of the roles separately.

The three main roles of metalanguages are: (1) *modeling* the variation space; (2) *implementing* the variability in the artifact; and (3) *selecting*, *generating*, or *configuring* a particular variant. The distinction between these roles is perhaps best understood in the context of feature-oriented software development (FOSD) [Apel and Kästner, 2009]. In FOSD, the basic unit of variation is a *feature*. One or many features can be combined with a *base program* in order to form a particular variant, called a *product*. The set of all products that can be generated is called a *product line*. In our terminology, a product line corresponds to a variational artifact, and a product to a single variant.

1. A variation *modeling* language describes the high-level constraints between features. For example, a common constraint is that the inclusion of one feature depends on the inclusion of another. The set of all such constraints in a product line is called a *feature model*. Features models can be expressed as diagrams [Kang et al., 1990], algebras [Höfner et al., 2006], propositional formulas [Batory, 2005], and more. The Kconfig language [Kconfig, She and Berger, 2010], used to organize configuration options in the Linux kernel, is a prominent example of variation modeling in a large-scale, real-world application [She and Berger, 2010, She et al., 2010].

2. A variation *implementation* language describes the realization of the features in terms of the object language. For example, CPP is a variation implementation language. To implement a feature f, introduce a new CPP variable name `F`, then wrap all object language code corresponding to f in CPP directives of the form `#ifdef F ... #endif`. Now feature f can be included by defining `F` when running the preprocessor. Note that if f depends on another feature g (realized by the CPP variable `G`), this constraint will likely not be expressed in the CPP-annotated source code. The implementation language of CPP describes only what object language code is associated with which feature, while a separate modeling language could be used to describe which combinations of features are valid.[1]

3. Finally, a *selection* or *configuration* language describes how to produce a particular variant from a variational artifact. In the simplest case, this may be just be a list of features to include. However, often the task of assembling a product is more complicated. In the compositional approach described in Section 2.3, the order that features are incorpo-

[1] Unfortunately, most actual CPP-annotated software projects do not provide corresponding feature/configuration models, making it difficult to determine which combinations of features are expected to produce complete, working programs.

rated is often significant, so the selection language must take this into account. In conjunction with CPP, the Make language [Make] can be considered a configuration language. A Make target corresponding to a single product would define the CPP variables corresponding to the product's features, and possibly include only the relevant source files as well.

In the choice calculus, the dimension declaration construct, defined in Section 3.3.2, is the primary mechanism for variation modeling. In the past we have considered selection to be external to the calculus itself, however, more recently we have explored language-level support for selection; this is the focus of Chapter 5. Variation implementation is achieved with the choice construct from which the calculus takes its name, discussed at length in Chapter 3.

2.3 Approaches to Representing Variation

In general, there are three ways to represent variation in software, which we will refer to as (1) *annotative*, (2) *compositional*, and (3) *metaprogramming-based*. The differences between these approaches are most pronounced when comparing the metalanguages used to implement variability and select individual variants. Often the same modeling languages, such as feature diagrams, are used in the context of multiple different approaches.

1. In the annotative approach, object language code is varied in-place through the use of a separate annotation metalanguage. Annotations delimit code that will be included or not in each program variant. When selecting a particular variant from an annotated program, the annotations and any code not associated with that variant are removed, producing a plain program in the object language. The most widely used annotative variation language is CPP, described in the previous section. An example in research is the CIDE tool, which associates

blocks of object language code with features through the use of background colors [Kästner et al., 2008a]. The choice calculus is principally annotative, though support for the other approaches described below are considered in Chapter 7.

2. The compositional approach emphasizes the separation of variational software into its component *features* and a shared *base program*, where a feature represents some functionality that may be included or not in a particular variant. Variants are generated by selectively applying a set of features to the base program. This strategy is usually used in the context of object-oriented languages and relies on language extensions to separate features that cannot be captured in traditional classes and subclasses. For example, inheritance might be supplemented by mixins [Bracha and Cook, 1990, Batory et al., 2004], aspects [Kiczales et al., 1997, Mezini and Ostermann, 2003], or both [Mezini and Ostermann, 2004]. Relationships between the features are described in separate configuration files [Batory et al., 2004], or in external documentation, such as a feature diagram. These determine the set of variants that can be produced, and may also describe how to assemble them.

3. The metaprogramming-based approach encodes variability using metaprogramming features of the object language itself. This is a common strategy in functional programming languages, such as MetaML [Taha and Sheard, 2000], and especially in languages in the Lisp family, such as Racket [Flatt and PLT, 2010]. In these languages, macros can be used to express variability that will be resolved statically, depending on how the macros are invoked and what they are applied to. Different variants can be produced by altering the usage and input to the macros.

Each of the three approaches to representing variation has its strengths and weaknesses. These are summarized in Figure 2.1. The qualities in the table are expressed positively (that is, "High" is always good), but they are not

Quality	Annotative	Comp.	Meta.
Object language independence	High	Low	None
Separation of concerns	None/Virtual	High	None
Variation explicitness	High	Medium	Low
Variation visibility	High	Low	Medium
Supports crosscutting variation	Low	High	Medium
Ease of adoption	High	Low	High

Figure 2.1: Summary of trade-offs between representation approaches.

weighted equally and their relative importance will vary depending on the user and task. Below is a brief summary of the meaning of each quality and why it matters to the design of a fundamental variation language and associated theory. A longer assessment of each approach in terms of these qualities follows.

- *Object language independence* indicates whether the ideas and tools associated with a strategy can be easily applied to many different artifact types. This is important in order to produce generalizable results that are not tied to a particular object language or usage context.

- *Separation of concerns* characterizes the modularity of the representation with respect to units of variation. For example, whether it can be used to encapsulate individual features and supports working on a single feature, independently of others. This is a generally useful quality that supports abstraction and scalability [Tarr et al., 1999].

- *Variation explicitness* indicates whether the representation of variability in an artifact is structured, precise, and straightforward to traverse and manipulate. This quality supports the analysis and transformation of variational structures.

- Relatedly, *variation visibility* indicates whether a representation makes it easy to determine which parts of the artifact are variational and how. This is important for the understandability of variational software.

- *Support for crosscutting variation* characterizes the ease of creating and maintaining variation that affects many different parts of an artifact in a similar way. The classic example of crosscutting variation in software is an optional logger, which may require extending every method in the program with an optional logging statement.
- Finally, *ease of adoption* indicates how easily a variation representation can be incorporated into an existing, non-variational artifact.

Although somewhat maligned in research [Spencer and Collyer, 1992, Pohl et al., 2005], we argue that the annotative approach achieves the best combination of these qualities for the goal of a general theory of variation, and is therefore a good basis for the choice calculus.

Annotative approaches have the highest degree of *language independence*. For example, CPP can be used with almost any textual object language, as long as its syntax does not interfere with the #-notation of CPP. Software projects usually consist of several different artifact types, such as source code, build scripts, documentation, etc. Language independence makes it easy to manage and synchronize variation across all of these different artifact types, and trivial to incorporate variation in new artifact types. Some compositional approaches, like the AHEAD tool suite [Batory et al., 2004], provide a degree of language independence through parameterization. By identifying an appropriate representation of a *refinement* for each object language type, and implementing an operation for composing refinements, the system can be extended to support new object languages. Metaprogramming approaches are usually tightly coupled to their object languages and so cannot be applied to other artifact types.

Of course, language independence comes at a cost. Since CPP knows nothing about the underlying object language, it is easy to construct variational programs in which not all variants are even syntactically correct, and this usually cannot be detected until the variant is generated and run through a compiler. This problem has been addressed in research by operating on the abstract syntax tree of the object language, rather than annotating the concrete syntax directly. This is the approach used by the annotative CIDE tool [Kästner et al., 2008a] and in the choice calculus [Erwig and Walkingshaw, 2011b], and will be discussed in greater detail in Section 3.3.1. This solution maintains a high degree of language independence while providing structure that ensures the syntactic correctness of all program variants.

Compositional approaches are strongly motivated by traditional software engineering pursuits such as *separation of concerns* (SoC) and *stepwise refinement* [Batory et al., 2003, 2004, Mezini and Ostermann, 2004, Prehofer, 1997]. These represent the ideals that the code corresponding to a feature should be in some way modular, understandable independently of other features, and able to be added to a software system without changing the existing source code. Neither annotative nor metaprogramming-based approaches directly support SoC, although Kästner and Apel [2009] propose the idea of a "virtual separation of concerns" (VSoC) where some of the benefits of SoC are brought to annotative systems by providing tool support for working with projections of annotated artifacts.

VSoC is only possible because annotative approaches provide an *explicit representation of variation* in an artifact. That is, variability is expressed in a precise and consistent way, revealing a variational structure that can be traversed and manipulated. This is an important but often overlooked feature of annotative variation since it supports the analysis and transformation of variability in the artifact. As can be seen in the list of contributions in Section 1.2, such analyses and transformations are a major theme of this thesis. Compositional approaches also provide a structured view of variation by encapsulating variable parts in modules, but the resulting variational structure is not as precise (often requiring significant redundancy) and usually more complicated than in annotative approaches. Metaprogramming-based approaches provide essentially no directly observable variation structure since the variants that can be generated are determined by arbitrary macro computations.

The trade-off in variation explicitness between annotative and compositional approaches reflects the fact that annotative approaches are *extensional* while compositional and metaprogramming-based approaches are *intensional*. That is, annotative approaches simply enumerate the differences between each variant, while the intensional approaches describe variation in terms of transformations (in the form of refinements, aspects, or macros) from a base program to a new program containing the desired set of features. While the intensional view is more powerful, the extensional view is simpler and easier to analyze. Even basic analyses like counting or enumerating variants, or ensuring that a variation implementation conforms to its model, are trivial in structured annotative representations,[2] but difficult or impossible in the other approaches.

Related to this is the issue of *variation visibility*, which is the ability to determine exactly which parts of an artifact are variational and the impact of that variation. This is difficult in metaprogramming-based approaches because it is not always clear which macros represent variational concerns. It is very difficult in compositional approaches since the addition of some kinds of components (such as aspects) can have far-reaching effects on existing, previously non-variational code. In contrast, annotations explicitly designate a part of the artifact as variational, and localize the effects of that variation.

Following from the intensional/extensional distinction, compositional approaches can provide better support for *crosscutting variation* than annotative representations. In FOSD, there are many examples of features that affect many different parts of a program in similar ways. For example, a security feature may require the insertion of code that performs an authentication check at the beginning of every sensitive method. Such use cases motivate

[2]Structured annotative representations include CIDE and the choice calculus, but not CPP [Kästner et al., 2008a, Erwig and Walkingshaw, 2011b].

transformative components, such as aspects, that provide the relevant code once and describe all of the points it should be inserted. In annotative approaches the variation associated with a crosscutting feature must be replicated at each of these points. Crosscutting variation can be implemented by metaprograms without the repetition of the annotative approach, but metaprogramming systems typically do not provide special support for this, which limits the benefit [Roychoudhury et al., 2003].

Finally, the simple extensional model of annotative variation, and the ability to use a single metalanguage with many different object languages, makes annotative approaches *easier to adopt* into existing non-variational software projects. When working with an object language with metaprogramming support, it is also easy to add variability (at least for one kind of artifact). In contrast, compositional approaches are more difficult to adopt since the software must be carefully structured in order to incorporate variation. Varying part of an existing program is likely to require refactoring.

While the annotative approach scores well on the majority of qualities discussed in this section, the two qualities it lacks—support for separation of concerns and crosscutting variation—are those most traditionally valued in software engineering research. Because of this there has been a tendency in research to value compositional approaches (which score well in these areas) over annotative approaches. However, annotative approaches remain extremely widely used in practice. For example, conditional compilation is used in essentially all large C projects, and as much as 22% of code in real C projects is made up of CPP directives [Ernst et al., 2002]. CPP is also used in large-scale Haskell projects, such as the Glasgow Haskell Compiler [GHC], and in many other contexts. For its widespread use alone, annotative variation is worthy of study.

More importantly, annotative approaches provide a good foundation for a formal language and general theory of variation. Our previous work on extending Hindley-Milner type inference to a variational lambda calculus

(VLC) illustrates the usefulness of annotative variation in the development of a non-trivial variational analysis [2012, 2013]. Many of the above qualities feature prominently in the type system for VLC. For example, object language independence allows us to use the same metalanguage for describing variation in both VLC expressions and in types, and the explicit variation structure enables type simplification during the inference process to improve efficiency. By providing a language independent, highly structured, and visible model of variation, annotative approaches best separate variational concerns from the object language and support the analysis and transformation of variational artifacts.

Chapter 3 – The Choice Calculus

This chapter will provide the core language and theory of the choice calculus. In order to be as general as possible and support the broadest range of applications, the core language is intentionally minimalistic and designed to be easily extended and instantiated for use in different contexts. In fact, the only essential construct in the choice calculus is the *choice*—all other language features can be optionally included or not in a particular instance of the choice calculus, making the representation extremely customizable.

Section 3.1 introduces the core concepts and constructs of the choice calculus, namely choices and their synchronization by dimensions of variation. The design of the language is motivated in Section 3.2, while Section 3.3 gives a formal definition of its syntax and a description of how this syntax can be extended and instantiated for different tasks. A denotational semantics for the choice calculus is defined in Section 3.4. The semantics definition is *compositional*, supporting extensibility in the language. In subsequent chapters we make use of this quality to add new features in the choice calculus without requiring major changes to the existing language definition. A compositional semantics definition is possible because the constructs in the choice calculus are highly *orthogonal*—that is, each construct does one thing, and constructs can be used in almost any combination. Orthogonality is a highly desirable quality in language design [Scott, 2009, p. 328].

However, not *every* syntactically generable term is a valid choice calculus expression. Section 3.3.3 defines a well-formedness condition for the basic choice calculus. This condition will be extended as we add new features to the language, culminating in a type system for the choice calculus presented in Chapter 5, where the type of a choice calculus expression encodes the potential decisions that expression represents.

The same variational program can be expressed in several different ways in the choice calculus. This is a useful quality since different representations are useful for different purposes. In Section 3.5 we define an equivalence relation for choice calculus expressions. This relation can be used to perform semantics-preserving transformations between choice calculus expressions. This relation will be extended with each new language feature introduced in this thesis.

Finally, in Section 3.6 we develop a design theory for choice calculus expressions, identifying semantic criteria to identify redundant choice calculus expressions and transformations to improve them.

3.1 Introduction to the Choice Calculus

In this section we will introduce the main concepts and constructs of the choice calculus. We use a running example of varying a simple program in the object language of Haskell, but the choice calculus is generic in the sense that it can be applied to any tree-structured document.

Consider the following four implementations of a Haskell function named `twice` that returns twice the value of its argument.

```
twice x = x+x        twice y = y+y
twice x = 2*x        twice y = 2*y
```

These definitions vary in two independent *dimensions* with two possibilities each. The first dimension of variation is in the name of the function's argument: the variants in the left column use `x` and those in the right column use `y`. The second dimension of variation is in the arithmetic operation used to implement the function: the variants in the top row use addition, those in the bottom use multiplication.

We can represent all four implementations of `twice` in a single choice calculus expression, as shown below.

$\textbf{dim}\ \mathit{Par}\langle x, y\rangle\ \textbf{in}$
$\textbf{dim}\ \mathit{Impl}\langle \mathit{plus}, \mathit{times}\rangle\ \textbf{in}$
$\texttt{twice}\ \mathit{Par}\langle\texttt{x},\texttt{y}\rangle\ \texttt{=}\ \mathit{Impl}\langle \mathit{Par}\langle\texttt{x},\texttt{y}\rangle\texttt{+}\mathit{Par}\langle\texttt{x},\texttt{y}\rangle,\texttt{2}*\mathit{Par}\langle\texttt{x},\texttt{y}\rangle\rangle$

The example begins by declaring the two dimensions of variation using the choice calculus **dim** construct. The syntax **dim** $\mathit{Par}\langle x, y\rangle$ declares a new dimension of variation *Par* with *tags* named x and y. The tags represent the possibilities in the dimension, in this case, the two possible parameter names. We can refer to the tag x in dimension *Par* as $\mathit{Par}.x$. This is called a *dimension-qualified tag*, or usually just a *qualified tag*. The **in** keyword introduces the scope of the dimension declaration, which extends to the end of the expression if not explicitly indicated otherwise (for example, by parentheses).

Each point of variation between the different implementations of `twice` is captured in a *choice* that is bound by one of the declared dimensions. For example, $\mathit{Par}\langle\texttt{x},\texttt{y}\rangle$ is a choice bound by the *Par* dimension with two *alternatives*, `x` and `y`. Note that `x` and `y` are terms in the object language of Haskell (indicated by `monospaced font`), while the tags x and y are identifiers in the metalanguage of choice calculus (indicated by *italics*).

Each dimension represents an incremental decision that must be made in order to resolve a choice calculus expression into a single program variant in the object language. The choices bound to a dimension are synchronized according to this decision. This incremental decision process is called *tag selection*. When we select a tag from a dimension, the corresponding alternative from every bound choice is also selected, and the dimension declaration itself is eliminated. For example, if we select the y tag from the *Par* dimension—that is, if we select the qualified tag $\mathit{Par}.y$—we would produce the following choice calculus expression in which the *Par* dimension has been eliminated and each of its choices has been replaced by its second alternative.

$\textbf{dim}\ \mathit{Impl}\langle \mathit{plus}, \mathit{times}\rangle\ \textbf{in}$
$\texttt{twice y =}\ \mathit{Impl}\langle\texttt{y+y},\texttt{2}*\texttt{y}\rangle$

If we then select *Impl.times*, we obtain the plain Haskell function below.

```
twice y = 2*y
```

In this way, we can select each of the four variant implementations of `twice` by making each possible combination of selections in the dimensions *Par* and *Impl*. This conceptual mapping from sequences of tag selections to plain object language variants is the basis for the formal semantics of the choice calculus defined in Section 3.4.

3.2 Design of a Variation Representation

Having introduced the main concepts of the choice calculus, in this section we motivate its design by discussing the rationale and significant design decisions we encountered along the way.

3.2.1 Semantics-Driven Design

Traditionally, the definition of a language proceeds from syntax to semantics. That is, first a syntax is defined, then a semantic model is decided upon, and finally the syntax is related to the semantic model [Felleisen et al., 2009, Fowler, 2005]. Elsewhere, however, we have proposed an inversion of this process, where the semantic domain of the language is identified first, then syntax is added incrementally and mapped onto this domain. We argue that this *semantics-driven* approach to language design leads to more principled, consistent, and extensible languages [Erwig and Walkingshaw, 2011a, 2012b]. In this subsection, we show how the semantics-driven approach motivates aspects of the design of the choice calculus.

The first step in the semantics-driven design process is to identify a semantic core that captures the *essence* of the domain as simply and generally as possible. This is obviously not a deterministic process, requiring insight and creativity by the language designers. For the domain of variational software, the fundamental operation seems to be selecting a program variant. That is, the essential property of a variational program is that, through some decision process, we can produce from it multiple different-but-related plain programs. Generalizing this idea of selection to arbitrary variational artifacts, we get as the basis for a denotational semantics a mapping from *decisions* to plain artifact *variants* represented in some object language.

The next step of the semantics-driven design process is to identify a minimal set of syntactic forms that support the creation of the denotations. While ultimately we want to support many different features, it is important to get this minimal core right first. An initial idea is to just represent the mapping from decisions to variants explicitly, in a single construct. For example, we might represent our `twice` example from the previous section as a choice between the four different variant definitions, each labeled with a unique name, as shown below.

$$\begin{array}{l}\langle \textit{x-plus}\colon \texttt{twice x = x+x},\ \ \textit{x-times}\colon \texttt{twice x = 2*x}, \\ \ \textit{y-plus}\colon \texttt{twice y = y+y},\ \ \textit{y-times}\colon \texttt{twice y = 2*y}\rangle\end{array}$$

However, this representation clearly misses a key aspect of our domain, which is that the we expect the variants of a variational artifact to be related, otherwise there would be no advantage to combining them in a single representation. This shows up in the above representation of the `twice` example by a high percentage of repeated code.

Therefore, we want to capture the variation between related artifacts *locally*, allowing us to reuse the shared context. Thus we need at least two constructs: one to represent localized variation points, and one to support the embedding of these variation points within a common context in the object language. In the choice calculus, we call these constructs *choices* and *structures*, respectively, and together they define the minimal core of the choice calculus (see Section 3.3).

This leaves us with several open language design questions. Recall that the basis of our semantics is a mapping from decisions to variants. One question is how to associate the decisions in this mapping with the resolution of individual variation points (choices). What does a variation point look like, and how does a decision resolve it into a non-variational expression? Should a single decision affect many variation points, and if so, how? These design questions are considered in Section 3.2.2. More fundamentally, we can ask how do we represent variation and what kinds of variability should a variation point express? This is the focus of Section 3.2.3.

3.2.2 Synchronizing Variation Points

As its name implies, the fundamental concept in the choice calculus is the *choice*. A choice captures a point of variation within an artifact by explicitly enumerating a list of *alternative* expressions, exactly one of which will be included in any particular program variant.

As seen in Section 3.1, we associate a dimension name with each choice in order to facilitate the resolution of choices into alternatives. When a tag is selected from a dimension, each bound choice is replaced by the alternative it contains that corresponds to the selected tag. This effectively *synchronizes* every choice in the same dimension.

However, we can imagine many alternative ways of resolving choices and organizing the variation space. One possibility is to simply let each choice stand on its own. Then each variation point represents an independent decision that must be made to produce a plain object language term from a variational expression. For example, our `twice` program from the previous section could be represented by stand-alone choices as follows.

```
twice ⟨x,y⟩ = ⟨⟨x,y⟩+⟨x,y⟩,2∗⟨x,y⟩⟩
```

Of course, this represents not just the four variants we originally intended, but many other variants of dubious merit, such as `twice x = 2∗y`, generated by selecting the first alternative in the first choice, and the second alternative in other choices until a term with no variation is achieved. Although in this case we can imagine factoring out the choices related to the parameter name and reusing them in some way (for example, see Chapter 4), this solution only works in the special case where all of the choices we want to synchronize have exactly the same alternatives. Therefore, it seems clear that we want some ability to synchronize the resolution of multiple choices.

Another possibility is to represent a choice as a direct mapping from tags to alternative expressions. Choice synchronization is then supported through overlapping choice domains—if the tag x is selected, every choice containing x in its domain is replaced by the corresponding alternative. For example, we can represent the `twice` program in this direct-tagging style as follows.

$$\texttt{twice}\ \langle x\colon \texttt{x}, y\colon \texttt{y}\rangle\ \texttt{=}\ \langle \mathit{plus}\colon \langle x\colon \texttt{x}, y\colon \texttt{y}\rangle \texttt{+} \langle x\colon \texttt{x}, y\colon \texttt{y}\rangle, \mathit{times}\colon \texttt{2}{*}\langle x\colon \texttt{x}, y\colon \texttt{y}\rangle\rangle$$

This implementation captures the exact same set of variants as the choice calculus representation from the previous section.

The direct tagging approach is quite flexible and expressive. For example, suppose we extend our `twice` example with a new function `thrice` that triples the value of its argument, whose implementation is *partially synchronized* with `twice`. (We also fix the parameter name to `x` for improved clarity.)

$$\begin{aligned}&\texttt{twice}\ \ \ \texttt{x = }\langle \mathit{plus}\colon \texttt{x+x}, \mathit{times}\colon \texttt{2*x}\rangle\\ &\texttt{thrice x = }\langle \mathit{times}\colon \texttt{3*x}, \mathit{twice}\colon \texttt{x+twice x}\rangle\end{aligned}$$

Observe that the implementations of the two functions will be synchronized if we select the *times* tag, but only `twice` supports the *plus* implementation, while `thrice` provides an implementation that reuses `twice`, tagged *twice*.

However, such partial synchronization is highly dependent on the order in which tags are selected. For example, observe that if we select either *plus* or *twice* first, this will eliminate one of the two choices; we can then select the *times* alternative in the remaining choice, revealing that the *times* alternatives are not really synchronized at all. Since different subsets of choices are eliminated by different tags, it is impossible, in general, to enforce that the selection of a tag will select every alternative associated with that tag. This makes it hard to ensure that even basic properties are satisfied by all program variants. This suggests that the direct-tagging approach is too unstructured to be a foundation for theoretical research.

Organizing tags into dimensions provides a simple solution. Dimensions are a bit like a type system for choices. By grouping the tags *plus* and *times* into a single dimension *Impl*, we say that every choice in dimension *Impl* must provide an alternative corresponding to each of those two tags. In other words, all choices that have to do with the same variational concern must have the same form. With this requirement, we no longer need to associate tags with alternatives directly, relying instead on each alternative's position within the choice.

3.2.3 Alternative vs. Optional Variation

An even more fundamental question is whether sets of alternative expressions are the best way to express variation points within an artifact. In Section 2.3 we have already made the case for extensional variation representations over intensional ones. That is, representations where the differences between variants are listed explicitly, rather than those based on transforming one variant into another. However, even within the extensional view there are many possible ways to capture these differences.

One alternative representation of variation points that must be considered is one based on *optionality* rather than choices between alternatives. For example, in the CIDE tool [Kästner et al., 2008a], variation points are indicated by colored blocks of code that can either be included or not in each variant. If a given feature is included in a variant, all blocks of the corresponding color are included, otherwise they are excluded.

Compared to alternative-based variation, optional variation is less expressive since only syntactically optional subexpressions can be made variational. For example, choosing C as an object language, statements and function definitions can be marked optional since omitting them will still yield a syntactically valid program, but many kinds of C expressions cannot be varied since omitting them will produce a syntax error. Alternatives subsume optionality as a special case. An optional expression is a choice between the expression and a (meta)syntactic marker in the object language that represents an "empty" expression. This enforces that optional variation occurs only where it is syntactically valid. We usually use the symbol $\circ$ to represent an empty expression, but a more concrete example might be the empty statement ; in the object language of C. For more on representing optional variation in the choice calculus, see Section 6.1.2.

Many optionality-based approaches support arbitrary *boolean inclusion conditions* on code that is marked as optional. For example, an optional statement s may be associated with an inclusion condition such $A \wedge (B \vee C)$, indicating that the statement should be included whenever features A and B *or* features A and C are included. Any such inclusion condition can be replicated in the choice calculus through choice nesting. For example, assuming the left alternatives of dimensions A, B, and C indicate that the corresponding feature is included, then $A\langle B\langle \mathsf{s}, C\langle \mathsf{s}, \circ\rangle\rangle, \circ\rangle$ represents a statement s with the inclusion condition above. However, complex conditions can require quite large choice structures with many redundancies. The sharing constructs in Chapter 4 can help, but ultimately, this represents a trade-off between the simple regularity and expressiveness of the choice calculus compared to the flexibility of arbitrary boolean inclusion conditions.

The implementation of choices by Kästner et al. [2011b] in TypeChef combines the expressiveness of alternative-based approaches with the flexibility of boolean inclusion conditions. The cost is a more complex choice structure. As in CPP, in TypeChef there are no dimensions structuring the tag space.

Instead, each alternative in a choice is labeled with a boolean combination of tags, and the labels are constructed such that for any valid selection of tags, exactly one label is true. This representation is maximally expressive but makes structural analyses and transformations more difficult. Once again, we prefer the simpler dimension-oriented structure, but there are advantages to each approach.

3.3 Syntax of the Choice Calculus

Having introduced the main concepts of the choice calculus and the rationale behind its design, in this section we define its syntax formally. This section also shows how the choice calculus can be instantiated with different object languages and extended with different language features to produce variant choice calculi tailored to the needs of specific tasks. In fact, the dimension declaration construct introduced in Section 3.1 is an optional extension to support local dimension scoping and the naming of alternatives in a dimension as tags. It can also be omitted, producing a choice calculus with global dimension scoping and numbered alternatives. We make use of such a variant to represent the types of variational programs in our work on variational typing [Chen et al., 2012, 2013]. Thus, the choice calculus presented in this chapter defines the kernel of an arbitrarily extensible family of languages for representing variation in tree-structured artifacts.

3.3.1 Representing the Object Language

The goal of the choice calculus is to provide a formal model to represent variation in all kinds of languages and documents. To this end, we employ a simple tree model to represent the object language of the underlying artifact. This allows us to focus on the variational aspects of the representation, while providing a general and structured model to build on.

$$\begin{array}{rcll} e & ::= & a\prec e,\ldots,e\succ & \textit{Object Structure} \\ & | & D\langle e,\ldots,e\rangle & \textit{Choice} \end{array}$$

Figure 3.1: Core choice calculus syntax.

When the object language is a programming language, this tree model corresponds to an abstract syntax tree. In the `twice` example, although we show the choice calculus notation embedded within the concrete syntax of Haskell, this is not a textual embedding in the way that, for example, CPP's `#ifdef` statements annotate arbitrary lines of text in a program's source code. Instead, choices and dimensions annotate the abstract syntax tree of the program. This imposes constraints on the placement and structure of choices and dimensions. For example, every alternative of a choice must be of the same syntactic category.

Although the underlying model always has this tree structure, whenever possible we render examples using concrete syntax for readability, as we did in Section 3.1. Sometimes, however, it is necessary to represent the underlying tree structure of the object language explicitly, which we do with Y-brackets. For example, we can render the AST for `twice x = x+x` as $\texttt{=}\prec\texttt{twice},\texttt{x},\texttt{+}\prec\texttt{x},\texttt{x}\succ\succ$, that is, the definition is represented as a tree that has the `=` operation at the root and three children, (1) the name of the function `twice`, (2) its parameter `x`, and (3) the RHS, which is represented by another tree with the operation `+` as a root, and its two arguments `x` and `x` as children.

More generally, an object language expression is given by some constant information a and a possibly empty list of subexpressions, written $a\prec e_1,\ldots,e_n\succ$. In this context, the term "expression" includes anything that can be represented in the object language. Note that we usually omit the brackets from the leaves of these expressions. So we write, for example, $\texttt{+}\prec\texttt{x},\texttt{x}\succ$ rather than $\texttt{+}\prec\texttt{x}\prec\succ,\texttt{x}\prec\succ\succ$, to explicitly represent the expression `x+x`.

$$\begin{array}{rcll} e & ::= & x & \textit{Variable} \\ & | & \lambda x.e & \textit{Abstraction} \\ & | & e\ e & \textit{Application} \\ & | & D\langle e,\ldots,e\rangle & \textit{Choice} \end{array}$$

Figure 3.2: Variational lambda calculus.

The most minimal version of the choice calculus is shown in Figure 3.1, consisting of the structure construct for representing object language terms and choices for representing variation points. All other versions of the choice calculus are either *instantiations* or *extensions* of this core language. Instantiations replace the generic structure constructs with more specific constructs related to a particular object language (see below). Meanwhile, extensions add new functionality to the choice calculus. For example, local dimension scoping can be added by the dimension declaration construct introduced in Section 3.1, which is the focus of the next subsection.

The core choice calculus can be viewed as a rose tree with two different types of nodes—choice nodes, which contain a dimension D, and structure nodes, which contain some information a, specific to the object language. This tree-view of choice calculus expressions is often useful. For example, we have used it to describe many aspects of our variational unification algorithm for inferring variational types in variational lambda calculus [Chen et al., 2012, 2013].

Instantiating the choice calculus amounts to replacing the generic tree structure construct, $a{\prec}e,\ldots,e{\succ}$, with the constructs of a particular object language. From another perspective, we can say that any object language can be made *variational* by incorporating the desired constructs of the choice calculus (at the minimum, choices). For example, Figure 3.2 shows the syntax of a variational lambda calculus. This language can be viewed as either the

$$\begin{array}{lcll} e & ::= & \varepsilon & \textit{Empty List} \\ & | & \textbf{cons}\ a\ e & \textit{Cons} \\ & | & D\langle e, \ldots, e \rangle & \textit{Choice} \end{array}$$

Figure 3.3: Variational lists.

core choice calculus instantiated by the untyped lambda calculus, or the untyped lambda calculus extended with choices.

The concept of instantiation is also a basis for variational data structures. Essentially, any data structure that can be represented by an algebraic data type can be made variational by simply extending the data type with a construct for choices. For example, Figure 3.3 gives the syntax of variational lists. With this language we can write expressions like $\textbf{cons}\ 1\ A\langle \textbf{cons}\ 2\ \varepsilon, \varepsilon \rangle$, which represents either the list `[1,2]` or the list `[1]`, depending on the selection in the dimension A.

Variational data structures have many applications. Obvious examples are to support tools and analyses related to variational software [Classen et al., 2011, Kenner et al., 2010]. but many more general examples are discussed in Chapter 7. Elsewhere, we have described the utility of variational graphs and data structures that support variational graph algorithms [Erwig et al., 2013b]. Although we mostly focus on variational software here, these applications emphasize the fact that the choice calculus is not just about software, but about identifying a general model of variation that applies in a broad range of contexts.

3.3.2 Structuring the Variation Space

As motivated in Section 3.2.2, choices are organized and synchronized through the use of dimensions of variation. It is important to separate the *concept* of a dimension from the dimension *declaration* construct **dim**, introduced in Section 3.1. While the dimension name associated with each

$$\begin{array}{rcll} e & ::= & a\prec e,\ldots,e\succ & \textit{Object Structure} \\ & | & \textbf{dim}\ D\langle t,\ldots,t\rangle\ \textbf{in}\ e & \textit{Dimension} \\ & | & D\langle e,\ldots,e\rangle & \textit{Choice} \end{array}$$

Figure 3.4: Choice calculus with local dimension declarations.

choice is a fundamental feature of the choice calculus, **dim** declarations can be considered an optional extension to the core choice calculus defined in the previous subsection (albeit one that has been a part of the language since the beginning [Erwig and Walkingshaw, 2011b]).

In the absence of local dimension declarations, we can assume that all dimensions in a choice calculus expression are *globally scoped*. This is significant because local dimension declarations can complicate many kinds of analyses. A motivation for globally scoped dimensions in the context of variational type inference is given in our previous work [Chen et al., 2013]. Alternatively, we can imagine many other kinds of dimension declaration and scoping mechanisms, aside from the **dim** construct. For example, we might associate dimensions with modules and allow more complex constraints between dimensions, as in the model by Kästner et al. [2012b].

With all of that said, throughout most of this thesis, we assume a choice calculus including local dimension declarations. This is not a limiting assumption since we can simulate global dimension scoping by simply declaring all dimensions once, at the top of an expression, then proceeding as if dimensions are globally scoped.

The syntax of the core choice calculus extended with the **dim** construct is given in Figure 3.4. This is the version of the choice calculus we will use in the rest of this chapter. The well-formedness property, denotational semantics, equivalence rules, and variational design theory are all defined in terms of this syntax. It is also the basis for the language extensions described in Chapters 4 and 5.

One useful feature of local dimension declarations is that a dimension can be *dependent* on a decision in another dimension. For example, consider the following three alternative implementations of `twice`, where the variants in the bottom row implement the function with a lambda expression, while the variant in the top row uses Haskell's operator section notation to define the function in a pointfree way (that is, without explicitly naming the variable).

```
twice = (2*)
twice = \x -> 2*x          twice = \y -> 2*y
```

Once again we have two dimensions of variation. We can choose a pointfree representation or not, and we can also choose the parameter name. In this case, however, it doesn't make sense to select a parameter name if we choose the pointfree style, because there is no parameter name! In other words, the parameter name dimension is only relevant if we choose "no" in the pointfree dimension. In the choice calculus, a dependent dimension is realized by nesting its declaration in an alternative of a choice in another dimension, as demonstrated below.

$$\begin{array}{l}\textbf{dim}\ \mathit{Pointfree}\langle \mathit{yes}, \mathit{no}\rangle\ \textbf{in}\\ \texttt{twice = }\mathit{Pointfree}\langle\texttt{(2*)}, \textbf{dim}\ \mathit{Par}\langle x, y\rangle\ \textbf{in}\ \texttt{\textbackslash}\mathit{Par}\langle\texttt{x},\texttt{y}\rangle\ \texttt{-> 2*}\mathit{Par}\langle\texttt{x},\texttt{y}\rangle\rangle\end{array}$$

If we select *Pointfree.yes*, we get the variant `twice = (2*)`, with no more selections to make. However, if we select *Pointfree.no* we must make a subsequent selection in the *Par* dimension in order to fully resolve the choice calculus expression into a particular variant.

In this way, the **dim** construct not only provides a scoping mechanism for dimension names, but also a way to do some basic variation modeling (see Section 2.2) by controlling which decisions are independent and which are dependent on a particular selection in another dimension.

$$\textsc{W-Obj}\quad \frac{\Gamma \vdash e_1\ \mathit{wf} \qquad \ldots \qquad \Gamma \vdash e_n\ \mathit{wf}}{\Gamma \vdash a{\prec}e_1, \ldots, e_n{\succ}\ \mathit{wf}}$$

$$\textsc{W-Dim}\quad \frac{\Gamma,\ D : n \vdash e\ \mathit{wf} \qquad n > 0}{\Gamma \vdash \mathbf{dim}\ D\langle t_1, \ldots, t_n\rangle\ \mathbf{in}\ e\ \mathit{wf}} \qquad\qquad \textsc{W-Chc}\quad \frac{D : n \in \Gamma \qquad \Gamma \vdash e_i\ \mathit{wf}}{\Gamma \vdash D\langle e_1, \ldots, e_n\rangle\ \mathit{wf}}$$

Figure 3.5: Well-formedness judgment.

3.3.3 Well Formedness and Other Static Properties

There are a few syntactic constraints on choice calculus expressions not captured by the syntax. (1) Each dimension must declare at least one tag. (2) All tags declared in a single dimension declaration must be pairwise different. (3) Each choice must be bound by a corresponding dimension declaration. (4) There must be exactly as many alternatives in a choice as there are tags in its binding dimension declaration. We call an expression that satisfies all of these constraints *well formed*.

The well-formedness property is defined by the judgment $\Gamma \vdash e\ \mathit{wf}$, which states that expression e is well formed in context Γ. The well-formedness judgment is defined in Figure 3.5. The context Γ maps dimension names to the number of tags/alternatives in that dimension. A structure node is well formed if each of its subexpressions is well formed. A dimension declaration is well formed if it defines at least one tag and its scope is well formed in the context extended with the newly declared dimension. Finally, a choice is well formed if it contains the appropriate number of alternatives and each of its subexpressions is well formed.

Using this judgment, we can say that an expression e is well formed if it is well formed in the empty context, that is, $\varnothing \vdash e\ \mathit{wf}$.

We can also refer to the dimensions that are *bound* or *free* in a given choice calculus expression. The bound dimensions in expression e, written $BD(e)$, are the set of dimension names that are declared in e. Meanwhile, the free dimensions in e, written $FD(e)$, are the set of dimension names that are referred to by choices in e that are not bound by a corresponding dimension declaration. This is similar to the standard definition of "free variables".

Note that it is possible for a single dimension name D to be both bound and free in the same expression. For example, in the expression $D\langle 1, \textbf{dim}\ D\langle t, u\rangle\ \textbf{in}\ 2\rangle$, dimension D is free since the choice referring to D is not bound by a corresponding dimension declaration, but D is also a bound dimension in D because of the dependent dimension declaration. In this case, the free dimension D is a *different* dimension than the bound dimension D, although there is no way to determine this from the name alone.

Figure 3.6 defines the functions $BD(\cdot)$ and $FD(\cdot)$ for computing the bound and free dimensions of a choice calculus expression, respectively. In the definitions, the notation x^n can be expanded to the sequence $x_1, \ldots, x_n$. The notation $i \in n$ means for every $i \in \{1, \ldots, n\}$. Both of these notations will be used throughout this thesis.

An expression e is called *dimension linear* if all bound dimensions in e are pairwise different. The rationale for this definition is that a dimension linear expression can be transformed into a useful normal form in which dimension declarations are maximally factored (see Section 3.5.3). Note that any expression can be made dimension linear by simply renaming conflicting dimensions and their bound choices.

Finally, it is often useful to talk about choice calculus expressions that do not include a particular syntactic category s. We say that such expressions are *s free*. For example, a choice-free expression does not contain any choices (but may still contain dimension declarations, structures, or any other syntactic categories. Additionally, we say that an expression is *variation free* if it is both dimension free and choice free. Finally, a choice calculus expression

$$\begin{aligned}
BD(a\text{≺≻}) &= \varnothing \\
BD(a\text{≺}e^n\text{≻}) &= \cup_{i \in n} BD(e_i) \\
BD(D\langle e^n \rangle) &= \cup_{i \in n} BD(e_i) \\
BD(\textbf{dim}\ D\langle t^n \rangle\ \textbf{in}\ e) &= \{D\}
\end{aligned}$$

$$\begin{aligned}
FD(a\text{≺≻}) &= \varnothing \\
FD(a\text{≺}e^n\text{≻}) &= \cup_{i \in n} FD(e_i) \\
FD(D\langle e^n \rangle) &= \{D\} \cup (\cup_{i \in n} FD(e_i)) \\
FD(\textbf{dim}\ D\langle t^n \rangle\ \textbf{in}\ e) &= FD(e) - \{D\}
\end{aligned}$$

Figure 3.6: Bound and free dimensions.

is considered *plain* if it contains only structure nodes—or only syntactic categories related to the object language, if the choice calculus has been instantiated by a particular object language. That is, a plain expression represents a non-variational term in the object language (e.g. a plain Java program). For now, all variation free expressions are also plain, but as we add new syntactic categories by various language extensions, this will no longer be the case.

3.4 Compositional Denotational Semantics

As established in Section 3.2.1, the semantics basis for the choice calculus is a *mapping* from *decisions* to *plain expressions*, that is, a mapping from decisions to the artifact variants that those decisions produce. Relating this to the syntax, we see that dimension declarations define the decisions that must be made, structure nodes describe pieces of the artifact that will be produced by

$$\begin{aligned}
[\![\textbf{dim } A\langle a,b\rangle \textbf{ in dim } B\langle c,d\rangle \textbf{ in } A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle]\!] = \\
\{([A.a, B.c], 1), ([A.a, B.d], 2), \\
([A.b, B.c], 3), ([A.b, B.d], 4)\}
\end{aligned}$$

$$\begin{aligned}
[\![\textbf{dim } C\langle e,f\rangle \textbf{ in } C\langle 1, \textbf{dim } D\langle g,h\rangle \textbf{ in } D\langle 2,3\rangle\rangle]\!] = \\
\{([C.e], 1), ([C.f, D.g], 2), ([C.f, D.h], 3)\}
\end{aligned}$$

Figure 3.7: Denotational semantics of some example expressions.

those decisions, and choices relate the two by associating alternatives with the decisions made about the dimensions.

Formally, a *decision* is represented by a sequence of *qualified tags*, where a qualified tag $D.t$ is a tag t prefixed by its dimension D. We use the metavariable q to range over qualified tags, δ to range over decisions, and ε to represent the empty decision containing no tags. Finally, we use adjacency, such as $q\delta$, to prepend a tag q to an existing decision δ, and to concatenate two decisions δ and δ', as $\delta\delta'$.

The variant corresponding to a particular decision can be obtained from a choice calculus expression through a process called *tag selection*. The order that tags are selected from an expression is determined by the order that dimension declarations are encountered during a pre-order traversal of the expression. For example, consider the two example expressions in Figure 3.7. The denotational semantics of each expression is given explicitly as a set of decision/plain-expression pairs. Observe in the first example that the tags in dimension A always appear before tags in dimension B since the declaration of A occurs before the declaration of B. Strictly ordering the tag selections reduces unnecessary selections and redundant entries in the semantics. This can be observed in the second expression, where the dimension D is dependent on the selection of $C.f$ (see Section 3.3.2). When

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

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

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

Figure 3.8: Definition of choice elimination.

selecting tag $C.e$, the D dimension declaration is eliminated before it is ever evaluated, so it does not appear in the decision of the first entry in the semantics. In the other cases, when $C.f$ is chosen, the declaration of dimension D remains, so a tag must also be selected from D to produce the variants 2 and 3.

Tag selection thus consists of (1) identifying the next dimension declaration, (2) selecting a tag, (3) eliminating the choices bound by that dimension, and then (4) eliminating the dimension declaration itself.

We call step (3) of the tag selection process *choice elimination* and define it as follows. Given a dimension declaration **dim** $D\langle t_1,\ldots,t_n\rangle$ and a selected tag t_i, we write $\lfloor e\rfloor_{D.i}$ to replace every free choice $D\langle e_1,\ldots,e_n\rangle$ in e with its ith alternative, e_i. A formal definition of choice elimination is given in Figure 3.8. The case for structure nodes, non-matching choices, and declarations of differently named dimensions just propagate the selection to their subexpressions. There are two interesting cases: First, recursion ceases if another declaration of dimension D is encountered, preserving local dimension scoping. Second, after a matching choice is replaced by its ith alternative e_i we recursively apply choice elimination to e_i. This means that we can nest choices in the same dimension, such as $D\langle D\langle 1,2\rangle,3\rangle$, and they will always both be eliminated by any selection in D. This makes it impossible

$$\begin{aligned}
[\![a\prec\succ]\!] &= \{(\varepsilon, a\prec\succ)\}\\
[\![a\prec e^n\succ]\!] &= \{(\delta^n, a\prec e'^n\succ) \mid ((\delta_i, e'_i) \in [\![e_i]\!])^{i:1..n}\}\\
[\![\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e]\!] &= \{(D.t_i\,\delta, e') \mid i \in \{1,\ldots,n\}, (\delta, e') \in [\![\lfloor e\rfloor_{D.i}]\!]\}
\end{aligned}$$

Figure 3.9: Denotational semantics of choice calculus expressions.

to select 2 from the nested choice above, so the second alternative of the inner choice is unreachable and can be considered dead. In our Section 3.6, we provide strategies for removing dead alternatives and other kinds of dead subexpressions.

Armed with choice elimination, we define the denotational semantics of the choice calculus in Figure 3.9. The semantics of a leaf node is trivial. For an internal structure node $a\prec e^n\succ$, we effectively compute the n-ary product of the semantics of the subexpressions by taking one pair (δ_i, e'_i) from each $[\![e_i]\!]$, then forming a new pair by concatenating each decision $\delta_1 \ldots \delta_n$ and building the resulting plain expression $a\prec e'_1, \ldots, e'_n\succ$. For a dimension declaration the semantic function recursively computes the semantics of the scope for each possible tag selection.

Note that there is no case for choices in the definition of $[\![\cdot]\!]$. This is because the semantics is defined only for well-formed expressions. If an expression e is well formed, then all of its choices are bound by a corresponding dimension declaration. Since the semantics of a dimension declaration recursively eliminates all of its bound choices, $[\![\cdot]\!]$ will never be invoked on a choice expression.

More generally, we express the fact that the semantics is defined for any well-formed choice calculus expression in the following lemma.

Lemma 3.4.1. *If $\varnothing \vdash e$ wf, then $[\![e]\!]$ is defined.*

This can be proved by induction on the structure of e using the argument above and the fact that the well-formedness property ensures that choice elimination will never be invoked on a choice with too few alternatives.

The important property of the semantics function is that it computes denotations that are mappings from decisions to plain variants.

Theorem 3.4.2. *If $\varnothing \vdash e$ wf, then $\forall e' \in rng(\llbracket e \rrbracket) : e'$ is plain.*

Proof. By induction on the structure of e. When $e = a\prec\succ$, the only variant is trivially plain. When $e = a\prec e^n \succ$, then for each resulting variant $a\prec e'^n \succ$, each subexpression e'_i is plain by the induction hypothesis, so $a\prec e'^n\succ$ is also plain. When $e = \textbf{dim } D\langle t^n \rangle \textbf{ in } e'$, the variability introduced by dimension D is eliminated in each variant, which is otherwise plain by the induction hypothesis. The case $e = D\langle e^n \rangle$ cannot occur by Lemma 3.4.1. □

The denotational semantics defined above is *compositional* in the sense that the denotation of each expression is constructed from the denotations of its subexpressions [Gunter, 1992, p. 21]. This is an important quality because it supports *modular language extensions*. For any language extension, assuming it can be mapped to the same semantics basis in a way that can also be expressed compositionally, we need only define the semantics of the new syntactic forms. Then we can reuse the semantics of existing forms unchanged. This is one of the main motivations behind semantics-driven language design (see Section 3.2.1), and it directly supports the view of the choice calculus as a family of languages where we can selectively choose which language features we want for a particular task or domain.

3.5 Semantics-Preserving Transformations

We can observe that the choice calculus representation is not unique, that choices can be represented at different levels of granularity, and that dimension definitions can be moved around too. For example, the three expressions

$$e = \textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ \texttt{5+}A\langle \texttt{1},\texttt{2}\rangle$$
$$e' = \textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ A\langle \texttt{5+1},\texttt{5+2}\rangle$$
$$e'' = \texttt{5+}\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ A\langle \texttt{1},\texttt{2}\rangle$$

Figure 3.10: Three semantically equivalent expressions.

in Figure 3.10 are all semantically equivalent, that is, $[\![e]\!] = [\![e']\!] = [\![e'']\!]$. This observation raises several questions: Does it matter which representation is chosen? Is one representation strictly better than another? Or do we need different representations and operations to transform between them?

It may be that different representations are useful for different purposes. For example, maximally factored choices (as in e and e'') isolate variability and maximize the sharing of common contexts as much as possible; this minimizes space requirements and avoids update anomalies during editing. On the other hand, fewer and bigger choices that repeat common parts may be better suited for comparing alternatives than many fine-grained choices. Moreover, having dimensions as close to the top of the expression as possible (as in e and e') reveals the variational structure better than deeply nested dimensions. This might be desirable or not, depending on the context.

In Section 3.5.1 we identify a complete set of equivalence rules between choice calculus expressions that can be used to transform expressions into a desired form without changing its semantics. As the choice calculus is extended with new language features in future chapters, this equivalence relation will be extended as well. If the choice calculus is instantiated by a particular object language, the rules can be similarly instantiated. We illustrate this in Section 3.5.2 by presenting the equivalence rules for the variational lambda calculus, whose syntax is given in Figure 3.2. Finally, in Section 3.5.3 we identify some useful normal forms and describe when and

Chc-Obj

$$D\langle a{\prec}e^n[i:e'_j]{\succ}^{j:1..k}\rangle \equiv a{\prec}e^n[i:D\langle e'^{j:1..k}\rangle]{\succ}$$

Chc-Dim

$$\frac{D \neq D'}{D\langle(\textbf{dim } D'\langle t^m\rangle \textbf{ in } e_i)^{i:1..n}\rangle \equiv \textbf{dim } D'\langle t^m\rangle \textbf{ in } D\langle e^n\rangle}$$

Chc-Chc-Swap

$$D\langle D'\langle [e^n[i:e'_j]]\rangle^{j:1..k}\rangle \equiv D'\langle e^n[i:D\langle e'^{j:1..k}\rangle]\rangle$$

Figure 3.11: Choice commutation rules.

how they can be attained by applying the equivalence rules as semantics-preserving transformations.

3.5.1 Equivalence Rules

A complete set of equivalence rules for the choice calculus can be obtained by observing that in principle any syntactic form, that is, *Object Structure*, *Choice*, or *Dimension*, can be commuted with any other. An attempt to systematically enumerate all possibilities reveals further that not only can we commute choices with other choices, we can also merge two choices in the same dimension. The naming of rules presented in this section are based on this enumeration, using a three-letter code for each syntactic construct being commuted (Obj, Chc, and Dim, respectively), with an optional suffix indicating further detail.

We present the rules in several groups. First, we show rules for factoring and distributing choices across other syntactic constructs in Figure 3.11. The rules make use of a notational convention to expose the *i*th element of a

$$\textsc{Chc-Idemp}\quad \frac{(e \equiv e_i)^{i:1..n}}{D\langle e^n\rangle \equiv e} \qquad\qquad \textsc{Chc-Chc-Merge}\quad D\langle e^n[i : D\langle e'^n\rangle]\rangle \equiv D\langle e^n[i : e'_i]\rangle$$

Figure 3.12: Choice simplification rules.

sequence. The pattern notation $e^n[i : e']$ expresses the requirement that e_i has the form given by the expression (or pattern) e'. For example, $e^n[i : e'\texttt{+1}]$ says that e_i must be an expression that matches $e'\texttt{+1}$, so the entire sequence e^n has the form $e_1, \ldots, e_{i-1}, e'\texttt{+1}, e_{i+1}, \ldots, e_n$.

When applied right to left (RL), the Chc-Obj rule lifts a choice out of the ith subexpression of a structure, repeating the previously shared context in each alternative. For example, given the expression $\texttt{1}{\prec}\texttt{2}, \texttt{3}, A\langle\texttt{4}, \texttt{5}, \texttt{6}\rangle{\succ}$, applying the Chc-Obj rule RL with $i = 3$ yields $A\langle\texttt{1}{\prec}\texttt{2}, \texttt{3}, \texttt{4}{\succ}, \texttt{1}{\prec}\texttt{2}, \texttt{3}, \texttt{5}{\succ}, \texttt{1}{\prec}\texttt{2}, \texttt{3}, \texttt{6}{\succ}\rangle$. Applied left to right (LR), the rule can be used to factor the shared parts of a structure out of a choice. In a similar way, the Chc-Chc-Swap rule applied RL lifts a choice out of the ith alternative of another choice (which may or may not be in the same dimension). However, when commuting choices and dimension declarations with rule Chc-Dim, we must ensure that they use different dimension names, otherwise the choice will escape (when applied RL) or be captured (when applied LR) by the dimension declaration, changing the semantics of the expression. Also note in the Chc-Dim rule that the same dimension declaration must occur in every alternative of the choice in order to be factored out. Otherwise D' is dependent on a subset of the tags in D and so cannot be factored out without changing the semantics.

We can also observe that some choices are semantically meaningless and can therefore be eliminated. The two choice simplification rules are given in Figure 3.12. When applied LR, the Chc-Idemp rule states that if every alternative of a choice is equivalent to some expression e, the choice can be replaced by e. Applied RL, it supports the introduction of a choice between equivalent alternatives.

The Chc-Chc-Merge rules reveal the property that outer choices *dominate* inner choices in the same dimension. For example, $D\langle D\langle 1, 2\rangle, 3\rangle$ is semantically equivalent to $D\langle 1, 3\rangle$ since the selection of the first alternative in the outer choice implies the selection of the first alternative in the inner choice. In this case, we say that the alternative 2 is *dead* since it can never be selected. When applied LR, the Chc-Chc-Merge rule eliminates a dominated choice by replacing it with the only live alternative e'_i. When applied RL the Chc-Chc-Merge introduces a dominated choice with arbitrary dead alternatives.

Having enumerated the commutation of choices with all constructs, we are left to consider the commutation of dimensions and structures. Note that we cannot commute two dimension declarations since their order is reflected in the semantics by the order that tags appear in decisions. We also cannot, in general, commute structures with structures since it might change the resulting variants. However, we *can* commute dimension declarations with structures, as long as the commutation does not alter the scope of the declaration or the ordering of dimensions. These conditions are reflected in the premises of the Dim-Obj equivalence rule in Figure 3.13. Considering the rule applied RL, the first premise ensures that lifting the dimension declaration out of the structure does not capture any free choices in the other subexpressions. The second premise states that if we lift the dimension out of the ith subexpression of the structure, then there can be no dimension declarations in subexpressions 1 through $i - 1$ since otherwise the ordering of dimension declarations (determined by a preorder traversal of the expression) would be altered. This means that in order to factor a dimension declaration out of the ith subexpression, we must first factor all dimension declarations out of the preceding subexpressions. This strategy is applied to achieve dimension normal form in Section 3.5.3.

$$\textsc{Dim-Obj}\quad \frac{D \notin \cup_{j \neq i} FD(e_j) \qquad (BD(e_j) = \varnothing)^{j:1..i-1}}{\textbf{dim}\ D\langle t^m \rangle\ \textbf{in}\ a{\prec}e^n[i:e]{\succ} \equiv a{\prec}e^n[i:\textbf{dim}\ D\langle t^m \rangle\ \textbf{in}\ e]{\succ}}$$

Figure 3.13: Dimension-structure commutation.

$$\textsc{Refl}\quad e \equiv e \qquad \textsc{Symm}\quad \frac{e \equiv e'}{e' \equiv e} \qquad \textsc{Trans}\quad \frac{e_1 \equiv e_2 \qquad e_2 \equiv e_3}{e_1 \equiv e_3} \qquad \textsc{Cong}\quad \frac{e \equiv e'}{\mathsf{C}[e] \equiv \mathsf{C}[e']}$$

Figure 3.14: Properties of the equivalence relation.

Finally, in Figure 3.14 we add reflexivity, symmetry, and transitivity rules to make $\equiv$ an equivalence relation. We also add a congruence rule to support transformations within a common context.

The defining feature of the equivalence relation is that equivalent expressions have the same semantics. That is, the equivalence relation is *sound* (though we do not claim that it is complete). Equivalently, we can say that the transformations defined by the the equivalence rules are *semantics preserving*. This quality is captured in the following theorem.

Theorem 3.5.1. *If* $\Gamma \vdash e$ *wf, then* $e \equiv e' \implies [\![e]\!] = [\![e']\!]$

A mechanized proof of this theorem for a simplified version of the choice calculus is given in Appendix A. A more traditional proof (that also includes proofs of the equivalence rules related to the **share** construct introduced in the next chapter) can be found in our previous work [Erwig and Walkingshaw, 2011b].

Chc-Abs

$$D\langle \lambda x.e_1, \ldots, \lambda x.e_n \rangle \equiv \lambda x.D\langle e_1, \ldots, e_n \rangle$$

Chc-App-l

$$D\langle e_1\ e', \ldots, e_n\ e' \rangle \equiv D\langle e_1, \ldots, e_n \rangle\ e'$$

Chc-App-r

$$D\langle e\ e'_1, \ldots, e\ e'_n \rangle \equiv e\ D\langle e'_1, \ldots, e'_n \rangle$$

Figure 3.15: Chc-Obj equivalence rule instantiated by lambda calculus.

3.5.2 Instantiating the Equivalence Rules

This section will briefly illustrate how the equivalence rules can be instantiated for a particular object language, demonstrating that the equivalence rules are generic with respect to the underlying structure of the artifact.

Recall that we instantiate the choice calculus syntax by replacing the structure construct with the constructs of the object language syntax. This was illustrated in Section 3.3.1 by instantiating the choice calculus by the lambda calculus. In this section we will show how choices and dimensions commute with lambda calculus syntax. This amounts to instantiating the Chc-Obj and Chc-Dim equivalence rules from Section 3.5.1 with the abstraction and application constructs of the lambda calculus (the variable reference syntactic form does not have any subexpressions, and so does not commute with choices or dimensions).

Figure 3.15 shows the instantiation of the Chc-Obj rule. Note that the instantiation produces three rules: one for the abstraction construct, and two for the application construct since we can commute with either the left or right subexpression of an application. The important quality of these rules is that they are *derived* from the Chc-Obj rule—no additional insight about the

Dim-Abs

$$\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ \lambda x.e \equiv \lambda x.\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e$$

Dim-App-l

$$\frac{D \notin FD(e')}{\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e\ e' \equiv (\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e)\ e'}$$

Dim-App-r

$$\frac{D \notin FD(e) \qquad BD(e) = \varnothing}{\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e\ e' \equiv e\ (\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e')}$$

Figure 3.16: Dim-Obj equivalence rule instantiated by lambda calculus.

object language is required. Conceptually, we can map each form on the LHS side of an instantiated equivalence rule to the generic structure representation of the uninstantiated choice calculus, set the parameter i according to the subexpression we want to commute with, perform the transformation using the Chc-Obj rule, then transform the result back into the object language to obtain the form on the RHS of the instantiated rule.

Figure 3.16 shows the instantiation of the Dim-Obj rule with the lambda calculus, which is derived from the Dim-Obj rule in the same way that the previous set of rules were derived from the Chc-Obj rule. Note that the premises in all three rules are simplified, relative to Dim-Obj, by the fact that in each case i is fixed and we have a known number of subexpressions. For example, in the rule Dim-Abs, since we only have one subexpression, the premises of Dim-Obj are trivially satisfied, so both can be omitted. Likewise, in the rule Dim-App-l, since $i = 1$ the second premise of Dim-Obj is trivially satisfied and thus omitted.

3.5.3 Dimension and Choice Normal Forms

The rules presented in Section 3.5.1 can be used to transform expressions in many different ways. In this section we identify three strategically significant representations: *dimension normal form*, *choice normal form*, and *dimension-choice normal form*. We then show that any expression can be transformed into choice normal form and that any dimension-linear expression can be transformed into dimension normal form and consequently, dimension-choice normal form.

We say that an expression e is in *choice normal form* (CNF) if it contains only choices that are maximally factored. That is, e is in CNF if no subexpression of e matches the LHS of any of the choice commutation rules in Figure 3.11, or the choice simplification rules in Figure 3.12, without violating a premise. CNF is significant because it minimizes redundancy in the representation.

Similarly, we say that an expression is in *dimension normal form* (DNF) if all dimensions are maximally factored. A dimension is maximally factored if its declaration appears at the top of the expression, at the top of an alternative within a choice, or directly beneath another maximally-factored dimension. DNF is convenient because it groups dimensions according to their dependencies. For example, all dimensions at the top of an expression are *independent*—the selection of any tag in an independent dimension does not affect the possible selections in other independent dimensions. Dimensions grouped within an alternative are *dependent* on the tag corresponding to that alternative being chosen—if the tag is not chosen, we need not make a selection in any dimensions in the group.

Finally, we say that an expression is in *dimension-choice normal form* (DCNF) if it is in choice normal form and in dimension normal form. DCNF combines the benefits of both CNF and DNF, avoiding redundancy and clearly revealing the dimension structure. We say that an expression is *linearly dimensioned* if it is both well formed and dimension linear (see Section 3.3.3). In the following we will show that any linearly dimensioned expression can be transformed into DCNF.

First, any expression e can be transformed into an equivalent expression e' that is maximally choice factored (in CNF). This can be achieved by repeatedly applying the rules in Figure 3.11 and Figure 3.12 from left to right.

Lemma 3.5.2. *$\forall e.\ \exists e'.\ e \equiv e' \wedge e'$ is in CNF.*

Proof. The definition of CNF is based on the applicability of the transformation rules. For an expression e, there are two possibilities: Either no rule can be applied, in which case e is in CNF already. Otherwise, a rule can be applied, which yields an expression e' to which the same reasoning can be applied inductively.

Now we must only show that this induction terminates. To support this, we define a measure, called the *choice height* of an expression e, as follows. Viewing e as a tree, the height of an individual choice is the length of the longest path from the choice to a leaf. Then the overall choice height of e is the sum of the heights of all choices. Now we can observe that the minimum choice height of an expression is 0, and that every rule Chc-* when applied LR strictly reduces the choice height. The rules, Chc-Chc-Swap, Chc-Chc-Merge, and Chc-Idemp all reduce the choice height by eliminating choices completely, while Chc-Obj and Chc-Dim reduce the choice height by moving a choice downward in the tree. Since the choice height of e is bounded and strictly decreasing, the transformation terminates. □

Lemma 3.5.2 is significant on its own, demonstrating that any expression can be transformed into *CNF*, minimizing redundancy. However, only expressions that are linearly dimensioned can, in general, be brought into dimension normal form.

Lemma 3.5.3. *If e is linearly dimensioned, then $\exists e'$ in DNF such that $e \equiv e'$.*

Proof. This result follows from the fact that we have a rule for moving a dimension out of each syntactic category. The premises that would prevent the application of a rule are of two forms: Either they (1) prevent the capture of free choices or they (2) constrain the order in which the rules can be applied. Premises of form (1) will never fail since e is well formed and also dimension linear. Premises of form (2) can be satisfied by factoring all dimensions out of left subexpressions before factoring dimensions out of right subexpressions. $\square$

From Lemma 3.5.2 and Lemma 3.5.3 the following result about dimension-choice normal form follows directly.

Theorem 3.5.4. *If e is linearly dimensioned, then $\exists e'$ in DCNF such that $e \equiv e'$.*

Recall the three example equivalent expressions from this section's lead, given in Figure 3.10. Comparing these to the definitions above, we see that e is in DCNF, while e' is (only) in DNF and e'' is (only) in CNF.

3.6 Variation Design Theory

We can observe that not every choice calculus expression is a good representation of variation. A trivial example is a choice of the form $D\langle e, e\rangle$ that contains two identical alternatives. Since it does not matter which alternative we select, the choice is superfluous and could simply be replaced by e.

In this section we will formalize several quality criteria for choice calculus expressions that can serve as guidelines for the design of "good" representations of variation. In Section 3.6.1, we describe the difference between syntactic and semantic approaches to deriving design criteria and motivate the semantic basis used throughout this section. In Section 3.6.2, we develop a semantic criterion for identifying equivalent alternatives in choices and equivalent tags in dimensions. We describe how to determine when these equivalent features are redundant and define transformations to remove them when they are. In Section 3.6.3, we develop a semantic criterion for identifying superfluous choices and dimensions and, again, transformations to remove them. Finally, in Section 3.6.4, we define a transformation for eliminating undesirable choice nesting patterns in order to remove unreachable alternatives.

3.6.1 Syntactic vs. Semantic Design Criteria

There are two ways to approach the formalization of variation design criteria. First, we can pursue a syntactic approach and identify patterns in choice calculus expressions directly, as we have done with the example $D\langle e, e\rangle$. However, it is not always syntactically obvious when two expressions are equivalent. Consider the following choice calculus expression *abc*.

$$\begin{array}{l} \textbf{dim}\ A\langle a, b\rangle\ \textbf{in} \\ \textbf{dim}\ B\langle c, d\rangle\ \textbf{in} \\ \textbf{dim}\ C\langle e, f\rangle\ \textbf{in} \\ A\langle B\langle C\langle 1,2\rangle, C\langle 3,4\rangle\rangle, C\langle B\langle 1,3\rangle, B\langle 2,4\rangle\rangle\rangle \end{array} \tag{abc}$$

In this example, the dimension A is superfluous—that is, the two alternatives of the only choice in A are equivalent, so it doesn't matter which tag in dimension A we choose—but this is difficult to see in the syntax. A proof is given in Figure 3.17 by transforming the right alternative of the choice in A into the left alternative through a sequence of applications of the semantics-preserving transformation rules given in Section 3.5. This transformation is described in detail below.

The high-level goal is to swap the nesting of the choices in dimensions B and C. To do this we will have to apply the Chc-Chc-Swap and Chc-Chc-Merge rules several times. We begin in step (1) by applying Chc-Chc-Swap with $i = 1$, to lift the first choice in B to the root of the expression. Now the two inner choices in B are dominated by the outer choice (see Section 3.5), but we cannot eliminate them straightaway since the only transformation rule that can eliminate a dominated choice is the

$$\begin{aligned}
C\langle B\langle 1,3\rangle, B\langle 2,4\rangle\rangle &\equiv B\langle C\langle 1, B\langle 2,4\rangle\rangle, C\langle 3, B\langle 2,4\rangle\rangle\rangle && (1)\\
&\equiv B\langle B\langle C\langle 1,2\rangle, C\langle 1,4\rangle\rangle, C\langle 3, B\langle 2,4\rangle\rangle\rangle && (2)\\
&\equiv B\langle B\langle C\langle 1,2\rangle, C\langle 1,4\rangle\rangle, B\langle C\langle 3,2\rangle, C\langle 3,4\rangle\rangle\rangle && (3)\\
&\equiv B\langle C\langle 1,2\rangle, B\langle C\langle 3,2\rangle, C\langle 3,4\rangle\rangle\rangle && (4)\\
&\equiv B\langle C\langle 1,2\rangle, C\langle 3,4\rangle\rangle && (5)
\end{aligned}$$

Figure 3.17: Proof that $C\langle B\langle 1,3\rangle, B\langle 2,4\rangle\rangle \equiv B\langle C\langle 1,2\rangle, C\langle 3,4\rangle\rangle$.

Chc-Chc-Merge rule, which requires the dominated choice to be nested directly within the dominating choice. Therefore, we must apply Chc-Chc-Swap twice more to setup our applications of Chc-Chc-Merge. In step (2), we apply Chc-Chc-Swap with $i = 2$ to the first alternative of the previous result, and in step (3), we do the same in the second alternative. Now the two inner choices in dimension B are nested directly beneath the outer choice in B, so we can apply Chc-Chc-Merge twice to eliminate them. In step (4) we apply Chc-Chc-Merge to the root with $i = 1$. In step (5) we do the same with $i = 2$.

Since the two alternatives of the choice in dimension A are equivalent, the expression *abc* is semantically isomorphic to the simple example $D\langle e, e\rangle$. Just as in the simple case, we can replace the choice in A with either of its alternatives without changing the meaning of the expression. However, unlike $D\langle e, e\rangle$, this fact is not syntactically obvious in the expression *abc*, as reflected by the non-trivial proof required to show that the two alternatives are equivalent.

As a different approach, we can formulate design criteria based on the *semantics* of choice calculus expressions. This means that we will attempt to recognize "bad" expressions not by looking at their syntactic forms, but by analyzing their semantics. This leads to a much simpler theory. For example, in expression *abc*, the replacement of the choice in A with one of its

$$\begin{aligned}[\![abc]\!] = \{\ & ([A.a, B.c, C.e], 1), && ([A.a, B.c, C.f], 2),\\ & ([A.a, B.d, C.e], 3), && ([A.a, B.d, C.f], 4),\\ & ([A.b, B.c, C.e], 1), && ([A.b, B.c, C.f], 2),\\ & ([A.b, B.d, C.e], 3), && ([A.b, B.d, C.f], 4) \quad \}\end{aligned}$$

Figure 3.18: Semantics of expression *abc*.

alternatives can be much more simply motivated by the fact that doing so does not change its semantics.

However, requiring semantic equivalence seems too strong a basis for an interesting design theory. With expression *abc* we can reasonably argue that since the only choice in A can be removed, any selection in A is irrelevant so the dimension itself should also be removed. The following expression *bc* is *not* semantically equivalent to *abc*, but it seems a valid simplification by the above argument.

$$\begin{aligned}&\textbf{dim}\ B\langle c, d\rangle\ \textbf{in}\\ &\textbf{dim}\ C\langle e, f\rangle\ \textbf{in}\\ &B\langle C\langle 1, 2\rangle, C\langle 3, 4\rangle\rangle\end{aligned} \tag{bc}$$

Compare the semantics of expression *bc*, given explicitly below, to the semantics of *abc* shown in Figure 3.18.

$$[\![bc]\!] = \{([B.c, C.e], 1), ([B.c, C.f], 2), ([B.d, C.e], 3), ([B.d, C.f], 4)\}$$

Observe that although the simplification from *abc* to *bc* is not semantics preserving, it is *variant preserving* in the following sense.

The *variants* of an expression e are the set of plain values that can be selected from it, obtainable from the denotational semantics as $rng([\![e]\!])$. Based on this, we say that two expressions e and e' are *variant equivalent*, written $e \sim e'$, if $rng([\![e]\!]) = rng([\![e']\!])$. And we call a transformation that maps e to e' *variant preserving* if $e \sim e'$.

Since the goal of variational software is ultimately to represent a set of related programs variants, variant equivalence seems a reasonable basis for a variation design theory. Using this, we see that $abc \sim bc$, since the variants represented by each expression are $\{1, 2, 3, 4\}$. So the transformation from abc to bc is variant preserving, and thus justified according to this criterion.

Compared to the syntactic approach, the semantic approach (based on variant preservation) provides a simpler and more general basis for a variation design theory. This is because it flattens the potentially complex structure of a choice calculus expression into a simple set of plain variants.

In the rest of this section we explore various redundancies that can occur in choice calculus expressions and describe the corresponding simplifications that can eliminate them. In Section 3.6.2, we investigate the binary equivalence of alternatives and tags. For equivalent tags, we define a variant-preserving transformation that reduces the size of the affected dimensions and choices. We explain why we do not do the same for equivalent alternatives. In Section 3.6.3, we lift the equivalence relations to sets of alternatives and tags, which correspond to choices and dimensions, and define variant-preserving transformations to eliminate these when needed as well.

3.6.2 Equivalent Alternatives and Tags

The simplest form of equivalence we can observe is between different alternatives in a single choice. We say that two alternatives e_i and e_j of a choice $D\langle e_1, \ldots, e_n\rangle$ are *equivalent* in context C, written $e_i \sim_C e_j$, iff the semantics of the choice is unchanged by swapping the alternatives e_i and e_j.

Definition 3.6.1 (Alternative equivalence)**.** $e_i \sim_C e_j \Leftrightarrow$
$[\![C[D\langle e^n[i : e_i, j : e_j]\rangle]]\!] = [\![C[D\langle e^n[i : e_j, j : e_i]\rangle]]\!]$

At first this definition may seem overly complicated. Why can't we just define alternative equivalence to be $[\![e_i]\!] = [\![e_j]\!]$? The reason is that we would like to be able to compare alternatives that, although part of a well-formed expression, are not themselves well-formed. For example, consider the choice $A\langle B\langle 1,2\rangle, B\langle 1,2\rangle\rangle$. The semantics of both alternatives, $[\![B\langle 1,2\rangle]\!]$, is undefined since B is unbound. But we would like to say that both alternatives are equivalent from a semantics perspective. Including a context C allows us to compare the semantics of expressions containing free dimensions. For $C[e_i]$ and $C[e_j]$ to be well-formed (and thus their semantics defined), C must declare all dimensions in $FD(e_i) \cup FD(e_j)$.

We define the function $\alpha_{C/i}(e)$ to perform the removal of the ith alternative of a choice in context C within expression e. This function is defined only if C is a context that matches a choice in e with at least i alternatives; that is, we assume $e = C[D\langle e_1, \ldots, e_n\rangle]$ with $n \geq i$. From this we can write a straightforward definition of alternative removal.

$$\alpha_{C/i}(e) = C[D\langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\rangle]$$

As an example, consider the following expression *abx*.

$$\textbf{dim}\ A\langle a,b,x\rangle\ \textbf{in}\ A\langle 1,1,9\rangle \qquad (abx)$$

Using the context $C = \textbf{dim}\ A\langle a,b,x\rangle\ \textbf{in}\ []$, we can remove the second alternative in the choice by applying $\alpha_{C/2}(abx)$. This yields the expression $\textbf{dim}\ A\langle a,b,x\rangle\ \textbf{in}\ A\langle 1,9\rangle$, which is not well formed since the dimension declares three tags but the choice contains only two alternatives.

The previous example demonstrates that we cannot simplify a choice with equivalent alternatives in isolation. Since the number of alternatives in a choice must match the number of tags in its binding dimension, reducing the number of alternatives in a choice requires removing the corresponding tag in the corresponding dimension. In this example, this would be fine since there is only one choice bound by dimension A. But in cases where we have other choices, the removal of the tag is possible only if all corresponding pairs of alternatives in all those other choices are also redundant.

On the dimension level, we can consider the equivalence of tags. As an example, consider the following expression *ab*.

$$\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ A\langle A\langle 1,1\rangle,1\rangle \tag{ab}$$

In this expression, the tags *a* and *b* are equivalent in the sense that selection with one tag yields the same result as selection with the other. Therefore, we define that two tags t_i and t_j are *equivalent* in context C, written $t_i \sim_C t_j$, iff the semantics of the expression is unchanged by swapping tags t_i and t_j.

Definition 3.6.2 (Tag equivalence). $t_i \sim_C t_j \Leftrightarrow$
$[\![C[\textbf{dim}\ D\langle t^n[i:t_i,j:t_j]\rangle\ \textbf{in}\ e]]\!] = [\![C[\textbf{dim}\ D\langle t^n[i:t_j,j:t_i]\rangle\ \textbf{in}\ e]]\!]$

In one sense, tag equivalence is a stronger property than equivalence of alternatives since a dimension that defines two equivalent tags can bind many choices, and thus the equivalence has a broader scope. However, equivalent tags do *not* imply that the corresponding alternatives of bound choices are also equivalent. This can be seen in the following simple example.

$$\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ A\langle A\langle 1,2\rangle,1\rangle \tag{ab'}$$

In this example, selection with either $A.a$ or $A.b$ produces 1 as a result. Therefore swapping the order of the tags would not impact the semantics, so tags *a* and *b* are equivalent. However, if we look at the two choices bound by the dimension A, neither pair of alternatives is equivalent, which we can express as $A\langle 1,2\rangle \not\sim_C 1$ and $1 \not\sim_C 2$.

Equivalent alternatives are not always a sign of fundamental redundancy. Often a dimension with more than two tags will have a choice in one part of the program with equivalent alternatives, but the corresponding alternatives will not be equivalent in other choices. As an example, consider a dimension for capturing differences related to the host operating system of a program: $\textbf{dim}\ OS\langle Linux, Mac, Windows\rangle$. Since Linux and Mac share a common heritage in Unix, there are bound to be some choices in this dimension where the first two alternatives are the same. For example, in a choice between newline characters, $OS\langle$`"\n"`$,$`"\n"`$,$`"\r\n"`$\rangle$, the first two alternatives are the same since Linux and Mac use the same newline character, but choices elsewhere, such as the location of a user's default home directory, will be different.

In contrast, equivalent tags are always redundant with respect to each other, and all but one of a set of a pairwise-equivalent tags can be removed while maintaining variant equivalence. We define the function $\tau_{C/u}(e)$ to perform the removal of tag u from the dimension declared at context C within expression e. Similar to alternative removal, tag removal is defined only if C is a context that matches a dimension declaration containing tag u. Assume $e = C[\textbf{dim}\ D\langle t^n[i:u]\rangle\ \textbf{in}\ e']$. To remove tag u, replace the dimension declaration with $\textbf{dim}\ D\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle$, in which tag u has been omitted, then replace every choice $D\langle e^n\rangle$ bound by the dimension with $D\langle e_1, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\rangle$, in which the ith alternative has also been omitted.

Applying tag removal to the equivalent tags in the examples in this section, we can perform the following variant-preserving transformations.

$$\begin{aligned}
\tau_{C/a}(abx) &= \textbf{dim}\ A\langle b, x\rangle\ \textbf{in}\ A\langle 1, 9\rangle & \tau_{C/b}(abx) &= \textbf{dim}\ A\langle a, x\rangle\ \textbf{in}\ A\langle 1, 9\rangle \\
\tau_{C/a}(ab) &= \textbf{dim}\ A\langle b\rangle\ \textbf{in}\ A\langle 1\rangle & \tau_{C/b}(ab) &= \textbf{dim}\ A\langle a\rangle\ \textbf{in}\ A\langle A\langle 1\rangle\rangle \\
\tau_{C/a}(ab') &= \textbf{dim}\ A\langle b\rangle\ \textbf{in}\ A\langle 1\rangle & \tau_{C/b}(ab') &= \textbf{dim}\ A\langle a\rangle\ \textbf{in}\ A\langle A\langle 1\rangle\rangle
\end{aligned}$$

The fact that removing a tag that is equivalent to another in the same context is always variant preserving is captured in the following theorem.

Theorem 3.6.3. $t_i \sim_C t_j \implies \tau_{C/t_j}(e) \sim e$

Proof. Following from Definition 3.6.2, we know that for every mapping $(\delta, e') \in [\![C[e]]\!]$ where $t_j \in \delta$, there exists another mapping to the same plain expression $(\delta', e') \in [\![C[e]]\!]$ where $t_i \in \delta'$, otherwise switching the order of t_i and t_j in the dimension declaration at C would have changed the semantics and $t_i \not\sim_C t_j$. Because of the existence of (δ', e'), we can remove the mapping

(δ, e') while preserving the variant e'. The function $\tau_{C/t_j}(e)$ just removes each such mapping (δ, e') and is therefore variant preserving when $t_i \sim_C t_j$. $\square$

Obviously, Theorem 3.6.3 applies to our examples *ab*, *ab'*, and *abx*. We can also apply it to the expression *abc* from the previous subsection, removing either tag a or b from dimension A. In all of these examples except for expression *abx*, the tag removal produces a dimension that contains only one tag and corresponding choices with only one alternative each. Such dimensions and choices are trivially superfluous and can therefore be eliminated. We will consider this kind of transformation in the next subsection.

3.6.3 Removing Pseudo-Choices and Pseudo-Dimensions

A choice in which all alternatives are pairwise equivalent is not really a choice at all since the decision of which alternative to pick has no impact on the semantics of the expression. We call a choice with this property a *pseudo-choice*. In the previous subsection we observed that a single pair of equivalent alternatives cannot be removed, in general. In contrast, a pseudo-choice can be safely replaced by any one of its alternatives. This fact is captured in the following lemma.

Lemma 3.6.4. $\forall i, j \in \{1, \ldots, n\} : e_i \sim_C e_j \implies [\![C[D\langle e^n \rangle]]\!] = [\![C[e_i]]\!]$

We define the function $\gamma_C(e)$ to replace the choice at context C within expression e with one of its alternatives. This operation is only defined when C matches a choice in expression e; that is, when $e = C[D\langle e^n \rangle]$. It is of course only semantics-preserving when the matched choice is a pseudo-choice.

As an example, consider again the expression *ab*. Since both alternatives $A\langle 1, 1 \rangle$ and 1 are equivalent (that is, $A\langle 1, 1 \rangle \sim_C 1$ with $C = \mathbf{dim}\ A\langle a, b \rangle\ \mathbf{in}\ []$), we can replace $A\langle A\langle 1, 1 \rangle, 1 \rangle$ by 1 (or $A\langle 1, 1 \rangle$) by applying $\gamma_C(ab)$. The same also applies to the expression *ab'*.

Assume in both expressions the application of γ replaces the choice with the simpler alternative 1, yielding $\mathbf{dim}\ A\langle a, b \rangle\ \mathbf{in}\ 1$. Since the dimension A

has no choices, removing it has no impact on the variants the expression represents. This fact is expressed more generally in the following lemma.

Lemma 3.6.5. $D \notin FD(e) \implies \textbf{dim}\ D\langle t^n \rangle\ \textbf{in}\ e \sim e$

Even more generally, we can extend the notion of tag equivalence to arrive at the notion of a *pseudo-dimension*, which is a dimension in which all tags are pairwise equivalent. Like pseudo-choices, pseudo-dimensions can be removed completely. Unlike with pseudo-choices, this transformation is not semantics preserving, but it is variant preserving.

The dimension A is a pseudo-dimension in both of the expressions ab and ab' since the only tags in dimension A, a and b, are equivalent. In contrast, the dimension A is not a pseudo-dimension in the expression abx since only tags a and b are equivalent, but neither is equivalent with tag x.

To support the elimination of pseudo-dimensions, we define a function $\psi_C(e)$ that completely removes the dimension D declared at context C in e. The function is defined only if C matches a declaration of dimension D, so we can assume that $e = C[\textbf{dim}\ D\langle t^n \rangle\ \textbf{in}\ e']$. To remove the dimension, we replace the dimension declaration with e', then replace each choice $D\langle e^n \rangle$ originally bound by D in e' with one of its alternatives e_i. We can reuse the choice elimination operation from Section 3.4 to perform this second step. Thus, we can define the dimension removal function as follows.

$$\psi_C(e) = C[\lfloor e' \rfloor_{D.1}] \qquad \text{where } e = C[\textbf{dim}\ D\langle t^n \rangle\ \textbf{in}\ e']$$

That fact that removing a pseudo-dimension in this way is variant preserving is captured in the following theorem.

Theorem 3.6.6. $\forall i, j \in \{1, \ldots, n\} : t_i \sim_C t_j \implies \psi_C(e) \sim e$

Proof. By induction over n. When $n = 1$, each choice bound by dimension D will have only one alternative. Since each such choice $D\langle e_1 \rangle$ trivially satisfies the condition of Lemma 3.6.4, it can be replaced by e_1. Subsequently there will be no choices bound by D, satisfying the condition of Lemma 3.6.5, so the dimension can also be eliminated. When $n > 1$, choose tags t_{n-1} and t_n. These satisfy the conditions of Theorem 3.6.3, so we can eliminate tag t_n by applying $\tau_{C/t_n}(e)$. Together these operations implement $\psi_C(e)$, and since each is at least variant preserving, $\psi_C(e)$ is also variant preserving. □

3.6.4 Removing Dominated Choices

Recall from Section 3.5 the Chc-Chc-Merge equivalence rule. When applied from right to left, this rule supports the removal of a dominated choice—that is, a choice that is nested within a choice in the same dimension. Since all but one alternative of a dominated choice can never be selected, dominated choices serve no real purpose, and the Chc-Chc-Merge rule seems a good basis for improving choice calculus expressions. However, since the rule is syntactic and can only be applied to *immediately* dominated choices, eliminating an arbitrary dominated choice can require many preparatory transformations. As illustrated in Section 3.6.1, this process can be quite tedious, which was the motivation for pursuing semantically-driven design criteria in the first place. So a natural question is: Can we express the idea of choice merging in a more declarative way, based on semantic, rather than syntactic, criteria?

A simple way to achieve this is by employing choice elimination, defined in Section 3.4. Within the ith alternative of a choice, we can safely project on the ith alternative of all nested choices in the same dimension. That is, if a choice $D\langle e^n\rangle$ is well formed in context C, then the following property holds.

$$[\![C[D\langle e^n[i:e_i]\rangle]]\!] = [\![C[D\langle e^n[i:\lfloor e_i\rfloor_{D.i}]\rangle]]\!]$$

This is because selecting the ith alternative in the outer choice necessarily implies the selection of the ith alternative in all nested choices in the same dimension. Generalizing this to project on all alternatives within a choice, we get the following lemma.

Lemma 3.6.7. $[\![C[D\langle e^n\rangle]]\!] = [\![C[D\langle \lfloor e_1\rfloor_{D.1}, \ldots, \lfloor e_n\rfloor_{D.n}\rangle]]\!]$

This lemma describes a form of algebraic optimization reminiscent of constant folding and propagation in compiler theory. Note also that this optimization is semantics-preserving, and so can be applied in contexts where we not only want to preserve the variants but the decisions a choice calculus expression represents.

The design criteria described in this section provide several ways to identify and eliminate incidental redundancy—that is, redundancy that serves no useful purpose in the representation of variational artifacts. However, some redundancy is essential and unavoidable using the choice calculus as presented so far. In the next chapter, we present two extensions to the choice calculus that support reuse in choice calculus expressions, enabling the elimination of many kinds of essential redundancy as well.

Chapter 4 – Extensions to Support Reuse

A truism of software engineering and programming languages is that any significant bit of functionality should be implemented only once, then reused wherever it is needed. This idea is known as the *abstraction principle* [Pierce, 2002, p. 339] or by the maxim "don't repeat yourself" [Hunt and Thomas, 1999]. To support this, the choice calculus has provided a mechanism for sharing since the beginning [Erwig and Walkingshaw, 2011b]. In subsequent work, however, we have encountered many subtle issues surrounding this feature. For example, the choice calculus as described in Chapter 3 has a monadic structure [Erwig and Walkingshaw, 2012a] (also see Section 7.1), but this is broken by the introduction of sharing. There are also some challenges and non-obvious design decisions regarding when and how shared expressions should be expanded, relative to dimensions and choices.

This chapter describes two different modular extensions to the choice calculus: One that supports the sharing of variant subexpressions, and one for the reuse of variational components. The distinction is mainly one of timing, that is, when a shared expression is expanded relative to the variational concerns expressed by choices and dimensions. This will be clarified in Section 4.2. These extensions can be independently included or not in a particular instance of the choice calculus, depending on which features and properties are needed in the metalanguage.

The rest of this chapter is structured as follows: Section 4.1 describes several scenarios that can benefit from sharing and reuse in the choice calculus and informally introduces the extensions to support these scenarios. Section 4.2 describes the fundamental staging distinction between the two extensions. The syntax of the extensions is defined in Section 4.3, and the well-formedness relation from Section 3.3.3 is extended correspondingly. The denotational semantics is extended in Section 4.4, and Section 4.5 enumerates the semantics-preserving transformations involving the extensions.

4.1 Motivation

There are at least four choice calculus usage scenarios that can benefit from language support for extracting and reusing subexpressions. These scenarios are grounded in our own use of the choice calculus and in similar issues encountered in related work, but they are intended to be more illustrative than comprehensive or typical. The scenarios are described briefly below, then in more depth in the rest of this section.

1. *Reuse of variation points.* A single dimension of variation can affect many places in an expression in a similar way. Rather than repeat the same choice over and over, we would like to extract and name the choice, then reuse it at each location.

2. *Reuse of alternatives.* In a dimension of three or more tags, there can be variation points where some (but not all) of the alternatives of a choice are the same. We would like to extract and name such repeated alternatives, then reuse them within the choice.

3. *Reuse of a common subexpression.* The variational part of an expression can "wrap" a common shared part—for example, an optional conditional statement around a shared code block. Rather than repeat the shared subexpression in both alternatives, we would like to extract and reuse it in each.

4. *Reuse of variational components.* The same variational component can occur in different parts of an expression. Once again, we would like to define this variational component once, then reuse it in each of these different contexts while retaining the ability to configure it differently at each place that it is reused.

4.1.1 Reuse of Variation Points

The running example from Section 3.1, of the four variants of the Haskell function `twice`, contains a typical example of our first reuse scenario. The choice calculus representation of this example is provided below, for reference.

> **dim** $Par\langle x, y\rangle$ **in**
> **dim** $Impl\langle plus, times\rangle$ **in**
> `twice` $Par\langle$`x`,`y`$\rangle$ `=` $Impl\langle Par\langle$`x`,`y`$\rangle$`+`$Par\langle$`x`,`y`$\rangle$,`2*`$Par\langle$`x`,`y`$\rangle\rangle$

Observe that the choice $Par\langle$`x`,`y`$\rangle$ is repeated four times. In fact, since the *Par* dimension captures the variability of the parameter name, we would expect *every* choice ever created in this dimension to be the same.

Although it is a minor issue in the case of parameter naming, the current representation of repeated choices is clearly *space inefficient*. More significant to this example, however, is that it is also *error prone*. The constraint that every choice should be exactly the same is not actually captured by the representation. If we want to change or extend the *Par* dimension, we must make the exact same edit in many different places. For example, suppose we extend our example with a third possible parameter name `z`, resulting in the following six variants.

```
twice x = x+x        twice y = y+y        twice z = z+z
twice x = 2*x        twice y = 2*y        twice z = 2*z
```

Implementing this extension requires several edits to the choice calculus expression. First we must extend the declaration of the *Par* dimension with a new tag *z*, then we must extend every choice in *Par* with a new alternative `z`. The resulting expression is shown below, with the newly added tag and alternatives underlined.

> **dim** $Par\langle x, y, \underline{z}\rangle$ **in**
> **dim** $Impl\langle plus, times\rangle$ **in**
> `twice` $Par\langle$`x`,`y`,$\underline{\texttt{z}}\rangle$ `=` $Impl\langle Par\langle$`x`,`y`,$\underline{\texttt{z}}\rangle$`+`$Par\langle$`x`,`y`,$\underline{\texttt{z}}\rangle$,`2*`$Par\langle$`x`,`y`,$\underline{\texttt{z}}\rangle\rangle$

Ideally, we could avoid such repetitive editing tasks. This problem, and the space efficiency issues, will only be exacerbated in larger, more complex choice calculus expressions.

The solution is of course to provide a mechanism that allows us to write the repeated choice once, name it, and share its result everywhere the parameter name is needed. For example, we can introduce a **share** construct that allows us to rewrite the above expression as follows.

dim $Par\langle x, y, \underline{z}\rangle$ **in**
dim $Impl\langle plus, times\rangle$ **in**
share v := $Par\langle$`x`, `y`, `z`$\rangle$ **in**
`twice` v `=` $Impl\langle v$`+`v, `2*`$v\rangle$

Now we need only add the alternative `z` in one place, and the result of the extended choice will be reused everywhere the choice calculus variable v is referenced in the definition of `twice`.

It is important to note that v is a variable in the metalanguage, not the object language. The binding symbol := is used to clearly distinguish choice calculus bindings at the metalanguage level from Haskell definitions at the object-language level.

4.1.2 Reuse of Alternatives

Another recurring scenario is that only some alternatives in a choice differ, while others are the same. For example, a program that varies depending on the choice of operating system—between Linux, Mac, and Windows—might have many choices in which the cases for Linux and Mac are the same since they share a common heritage in Unix. It would be inconvenient, error prone, and inefficient to duplicate the common code in each of these cases.

We can use the same **share** construct introduced in the previous section to solve this problem as well. Consider the following example of a simple choice calculus expression with a duplicated alternative (repeated from

```
newline =
#if LINUX || MAC
          "\n"
#elif WINDOWS
          "\r\n"
#endif
```

Figure 4.1: Alternative reuse in CPP.

Section 3.6.2). The expression is a variational Haskell program defining the newline character, which is the same on Linux and Mac (`"\n"`).

$$\textbf{dim}\ \mathit{OS}\langle \mathit{Linux}, \mathit{Mac}, \mathit{Windows}\rangle\ \textbf{in}$$
$$\texttt{newline = } \mathit{OS}\langle \texttt{"\textbackslash n"}, \texttt{"\textbackslash n"}, \texttt{"\textbackslash r\textbackslash n"}\rangle$$

Using the **share** construct, the repeated alternative can be extracted and named, then referenced twice in the choice.

$$\textbf{dim}\ \mathit{OS}\langle \mathit{Linux}, \mathit{Mac}, \mathit{Windows}\rangle\ \textbf{in}$$
$$\textbf{share}\ v := \texttt{"\textbackslash n"}\ \textbf{in}$$
$$\texttt{newline = } \mathit{OS}\langle v, v, \texttt{"\textbackslash r\textbackslash n"}\rangle$$

As with repeated choices, the advantages of sharing repeated alternatives grow as programs get larger and more complex.

As a technical aside, the need for this kind of sharing is a byproduct of the choice calculus's somewhat rigid alternative-based approach to representing variation points, described in Section 3.2.3. Approaches based on arbitrary boolean inclusion conditions tend to be more flexible. For instance, the example above can be represented in CPP without duplication or sharing, as shown in Figure 4.1.

This illustrates a trade-off between conditional and alternative-based approaches. On the one hand, alternative-based approaches provide a simpler and more regular structure, and are usually more expressive since they are not limited to varying syntactically optional expressions. On the other hand, conditional approaches are more flexible in the conditions they can express on optionally included code and so can avoid certain kinds of code duplication related to variability.

However, of the four motivating use cases presented in this section, only the reuse of alternatives is subsumed by arbitrary boolean inclusion conditions, so it does not remove the need for sharing in general. The fact that sharing can also express this use case without the more flexible conditional representation provides an orthogonality argument for the simpler alternative-based representation of variation points since it better decouples the issues of sharing and variation.

4.1.3 Reuse of Common Subexpressions

The previous use case motivating the reuse of alternatives can be viewed as an instance of a more general problem of reusing arbitrary subexpressions.

Repeated code can arise in all kinds of contexts that have nothing to do with variability. For example, if we're implementing a physical simulation, we might have code related to collision detection in many places throughout the system. Usually we would eliminate this kind of repetition using standard software engineering techniques, such as extracting the repeated code out into its own function or method. Of course, these cases are not the purview of a variation language.

Other kinds of repetition are a direct result of variation, however. One example is the *optional wrapper* problem described by Kästner et al. [2008b]. An optional wrapper is a variation pattern where the goal is to conditionally wrap an expression in another construct, such as a conditional statement or an exception handling construct.

For example, suppose there is a Java method, `readFile` that reads the contents of a file into a string but will fail with a `PermissionDenied` exception if the user does not have permission to read the file. The signature of this method is shown below.

```
String readFile(File file) throws PermissionDenied
```

In one variant of our program, we simply call `readFile` without any exception handling, letting a `PermissionDenied` exception propagate up the call chain. The result is stored in a `String` variable named `contents` defined elsewhere. We'll refer to this snippet of object language code as *read*.

```
contents = readFile(new File("TopSecret.txt"));
```
(*read*)

In another variant we incorporate an extremely aggressive security feature, initiating a self-destruct sequence if the user tried to access a file they don't have access to.

```
try {
  contents = readFile(new File("TopSecret.txt"));
} catch (PermissionDenied e) {
  System.selfDestruct();
}
```

How do we represent both of these variants in a single variational program? Using CPP, this is trivial, we just conditionally include the lines corresponding to the try-catch block if the security feature is included and exclude it otherwise, as shown in Figure 4.2.

However, this kind of code—where conditionally included lines of text do not correspond to whole statements or subexpressions—is exceptionally error-prone since it ignores the underlying structure of the source code [Spencer and Collyer, 1992, Liebig et al., 2011]. Recall from Section 3.3.1 that the choice calculus and other structured annotative representations, such as CIDE [Kästner et al., 2008a], instead annotate the abstract syntax tree of the object language. This avoids many of the problems with the kind of text

```
#ifdef SECURE
try {
#endif
  contents = readFile(new File("TopSecret.txt"));
#ifdef SECURE
} catch (PermissionDenied e) {
  System.selfDestruct();
}
#endif
```

Figure 4.2: Optional wrapper in CPP.

munging supported by CPP, guaranteeing, for example, that every variant that can be generated is at least syntactically correct.

So how would we represent this variational program in the choice calculus? Since, in the abstract syntax tree, *read* is a child of the try-catch block in the second variant, the only way to represent it without sharing is by duplicating *read*, as shown in abstract form below.

$\mathbf{dim}\ \mathit{Secure}\langle \mathit{no}, \mathit{yes}\rangle\ \mathbf{in}$
$\mathit{Secure}\langle$`contents = ...`$,$`try { contents = ...  } catch ...`$\rangle$

This kind of repetition is undesirable for the same reasons as the previous use cases—it is inefficient and error prone.

Since this is a common pattern, CIDE provides special support for optional wrappers [Kästner et al., 2008b]. Certain kinds of nodes in the syntax tree can be marked as wrappers, which allows their children to be marked as non-optional, even if the wrapper itself is optional. Therefore, in CIDE, starting from the second variant, the try-catch block would be marked as optional and associated with the *Secure* feature. Then the `contents ...` line within this optional block would be marked as *non*-optional.

In the choice calculus, we can factor out the redundancy with sharing, as shown below.

$\textbf{dim}\ \mathit{Secure}\langle \mathit{no}, \mathit{yes}\rangle\ \textbf{in}$
$\textbf{share}\ \mathit{read} := \texttt{contents = readLine("TopSecret.txt");}\ \textbf{in}$
$\mathit{Secure}\langle \mathit{read}, \texttt{try \{ } \mathit{read} \texttt{ \} catch ...}\rangle$

This again illustrates the orthogonality of sharing in the choice calculus. Using the same **share** construct we used to address the previous use cases, we are also able to solve the optional wrapper problem, which requires special support in other variation languages.

4.1.4 Reuse of Variational Components

The final kind of reuse we would like to support is the reuse of *variational components*. That is, we would like to define a subexpression containing variation once, then reuse it in such a way that it can be *configured differently* in each place.

For example, consider the following variational Haskell program that defines a variational function `sort` that sorts an arbitrary list of comparable elements, then uses that function in multiple places later in the program. The `sort` function varies in terms of the sorting algorithm it is implemented by.

```
sort = dim SortAlg⟨select,merge,quick⟩ in
       SortAlg⟨selectSort,mergeSort,quickSort⟩
minimum = head . sort
maximum = head . reverse . sort
```

Observe that we can make a selection in the *SortAlg* dimension only once and that decision will be reflected at each point that `sort` is used. For example, if we select *SortAlg.merge*, the definition of `sort` will be fixed to `mergeSort`, so every use of `sort` throughout our program will use the merge sort algorithm.

$\textbf{macro}\ \mathit{sort} := \textbf{dim}\ \mathit{SortAlg}\langle \mathit{select}, \mathit{merge}, \mathit{quick}\rangle\ \textbf{in}$
$\qquad \mathit{SortAlg}\langle \texttt{selectSort}, \texttt{mergeSort}, \texttt{quickSort}\rangle$
$\textbf{in}$
$\texttt{minimum = head .}\ \mathit{sort}$
$\texttt{maximum = head . reverse .}\ \mathit{sort}$

Figure 4.3: Reuse of a variational component using **macro**.

However, different sorting algorithms have different qualities, so we might want to select independently which sorting algorithm is employed at each use of the `sort` function. In our example, selection sort will probably be the fastest algorithm for the definition of the `minimum` function. Since Haskell only lazily sorts the list as each element is needed, `minimum` is $O(n)$ with selection sort since it must only "sort" one element to find the first/smallest element in the list. For the (admittedly silly) definition of `maximum`, however, a more sophisticated algorithm like merge sort will perform better since the use of `reverse` forces the whole list to be sorted. For still other cases, we might prefer quick sort over merge sort, for example, if we don't care about stability and expect the input to be random.

In other words, what we want is the ability to reuse variational expressions in a way that *preserves the variability* within those expressions. In the choice calculus, this amounts to duplicating the corresponding dimension declarations at each point of reuse. Each duplicated dimension will require a separate selection in order to resolve the variational program into a single program variant.

For this kind of reuse, we introduce a new construct **macro**. The distinction between **share** and **macro** is not yet clear but will be motivated in the next section. Using the **macro** construct we can get the desired behavior by replacing the Haskell `sort` function with a choice calculus macro *sort*, as shown in Figure 4.3. The variational program in the figure is exactly equivalent to the following one, in which the *sort* macro has been textually expanded into the definitions of the `minimum` and `maximum` functions.

```
minimum = head . dim SortAlg⟨select, merge, quick⟩ in
                 SortAlg⟨selectSort,mergeSort,quickSort⟩
maximum = head . reverse . dim SortAlg⟨select, merge, quick⟩ in
                 SortAlg⟨selectSort,mergeSort,quickSort⟩
```

However, in Section 4.4 we'll see that the semantics of **macro** are a bit more subtle than a simple textual expansion since we want to preserve the lexical scope of free choices within a macro definition.

4.2 Staging Relative to Dimension Elimination

Let us return again to the `twice` example from Section 4.1.1. Consider the case where we have selected *Impl.plus*, leaving us with the following variational program with three variants.

```
dim Par⟨x, y, z⟩ in
share v := Par⟨x,y,z⟩ in
twice v = v+v
```

Notice that although we have successfully extracted and reused the choice of the parameter name, there is nothing preventing us from creating other choices in the dimension *Par*. In Figure 4.4.a we extend our program with an additional function `thrice` that triples the value of its argument. This function also varies in terms of its parameter name by introducing a new choice in the *Par* dimension. Since both choices are in the same *Par* dimension, they will be synchronized. So, for example, whenever parameter name `y` is used in `twice`, the corresponding parameter name `b` will be used in `thrice`. Since `y` and `b` are in no way related, this may not be what we intended.

If we want the parameter names of the two functions to vary independently, then we must declare a new parameter name dimension for the new

```
dim Par⟨x,y,z⟩ in
share v := Par⟨x,y,z⟩ in
twice v = v+v
share w := Par⟨a,b,c⟩ in
thrice w = w+w+w
```

a. Synchronized choices.

```
dim Par⟨x,y,z⟩ in
share v := Par⟨x,y,z⟩ in
twice v = v+v
dim Par⟨a,b,c⟩ in
share w := Par⟨a,b,c⟩ in
thrice w = w+w+w
```

b. Independent choices.

```
share v := dim Par⟨x,y,z⟩ in Par⟨x,y,z⟩ in
twice v = v+v
share w := dim Par⟨a,b,c⟩ in Par⟨a,b,c⟩ in
thrice w = w+w+w
```

c. Independent, atomic choices.

Figure 4.4: Varying the parameter names of `twice` and `thrice`.

function `thrice`. One way to do this is to simply introduce a new dimension declaration before the second **share**-expression, as shown in Figure 4.4.b.

While the example in Figure 4.4.a is not incorrect, it is easy to see how this situation is potentially error prone. The assumption when *Par* is declared is that it will only ever be used in the single choice *Par*⟨x, y, z⟩, but its scope is actually much larger. As a solution, we can enforce the assumption using the **share** construct by pushing the declaration of *Par* into the bound expression, as shown in Figure 4.4.c. Now we cannot introduce new choices in the *Par* dimension since the scope of the dimension declaration is precisely the single choice associated with it. We call such single-choice dimensions *atomic*, a concept that will be revisited in Section 7.1.

This example reveals the fundamental distinction between the **share** and **macro** constructs introduced in the previous section. While **macro** supports the reuse of a variational expression in a way that lets us configure it differently at each use, **share** allows us to reuse the *result* of configuring an expression. Had we used **macro** instead of **share** in the example in Figure 4.4.c, we would have seven separate selections to make—three corresponding to each reference of v and four corresponding to each reference of w. Obviously, this is not what we want since we want to ensure that all uses of a parameter name in a single function are synchronized. Since the bound expressions in Figure 4.4.a and Figure 4.4.b contain no dimension declarations, their meaning would be unchanged by using **macro**.

The distinction between the two extensions can be more precisely understood in terms of *staging* [Sheard, 2001] relative to the elimination of dimensions by tag selection (see Section 3.4): **macro** constructs are expanded before dimension elimination (at an earlier stage), while **share** constructs are expanded after dimension elimination (at a later stage). In previous work we have experimented with both stage orderings. In the original choice calculus paper we use late-stage sharing [2011b], while in more recent work we use early-stage sharing [2013a]. Confusingly, we have used the keyword **let** for both forms! Since there are distinct use cases for each, we present both extensions here. The choice calculus can be extended with either or both forms of reuse, as needed.

The staging view also reveals that **macro** and **share** form a complete set of static reuse mechanisms with respect to dimension elimination since a reused expression must be expanded either before or after dimension elimination. However, in Chapter 6 we explore a more dynamic approach to reuse through function abstraction and application.

4.3 Syntax and Well-Formedness

The syntax of the **share** and **macro** extensions is given in Figure 4.5. Note that we use the same variable namespace for both extensions, and therefore have only a single syntactic form for variable references. This form must

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \textbf{share}\ v := e\ \textbf{in}\ e & \textit{Sharing Definition} \\ & | & \textbf{macro}\ v := e\ \textbf{in}\ e & \textit{Macro Definition} \\ & | & v & \textit{Variable Reference} \end{array}$$

Figure 4.5: Syntax of share and macro extensions.

$$\begin{aligned} BD(\textbf{share}\ v := e\ \textbf{in}\ e') &= BD(e) \cup BD(e') \\ BD(\textbf{macro}\ v := e\ \textbf{in}\ e') &= BD(e) \cup BD(e') \\ BD(v) &= \varnothing \end{aligned}$$

$$\begin{aligned} FD(\textbf{share}\ v := e\ \textbf{in}\ e') &= FD(e) \cup FD(e') \\ FD(\textbf{macro}\ v := e\ \textbf{in}\ e') &= FD(e) \cup FD(e') \\ FD(v) &= \varnothing \end{aligned}$$

Figure 4.6: Extension of bound and free dimensions.

obviously be included in any variant of the choice calculus that includes at least one of the two extensions.

For both **share** and **macro** expressions, we refer to the first subexpression as the *bound* expression and the second as the *scope*. For example, in the expression **share** $v := e$ **in** e' the expression e is bound to v in scope e'.

Recall from Section 3.3 that an expression is considered variation free if it is choice free and dimension free. Additionally, we say that an expressions is *reuse free* if it is share free, macro free, and reference free. Since a plain expression corresponds to a plain term in the object language, a plain expression is both variation free and reuse free.

We extend the definitions of bound and free dimensions to the new constructs in the obvious way, as shown in Figure 4.6. Similarly, we can refer

$$
\begin{aligned}
FV(a\prec\succ) &= \varnothing \\
FV(a\prec e^n\succ) &= \cup_{i\in n} FV(e_i) \\
FV(D\langle e^n\rangle) &= \cup_{i\in n} FV(e_i) \\
FV(\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e) &= FV(e) \\
FV(\textbf{share}\ v\ := e\ \textbf{in}\ e') &= FV(e) \cup (FV(e') - \{v\}) \\
FV(\textbf{macro}\ v := e\ \textbf{in}\ e') &= FV(e) \cup (FV(e') - \{v\}) \\
FV(v) &= \{v\}
\end{aligned}
$$

Figure 4.7: Free variables.

to the free and bound variables in an extended choice calculus expression. We use $FV(e)$, defined in Figure 4.7, to denote the set of free variables in e. Note that this definition reflects the fact that neither extension is recursive—that is, in an expression **share** $v\ := e$ **in** e', we cannot refer to v in e.

Finally, we extend the well-formedness property from Section 3.3.3 by adding the additional requirement that a well-formed expression contains no free variables. The extension to the well-formedness judgment is shown in Figure 4.8. Note that we reuse the environment Γ to keep track of which variables are in scope. For **share** and **macro** expressions we require that the bound expression is well formed and that the scope is well formed in the context extended by the new variable. Then a variable reference is well formed if it is in the context.

4.4 Denotational Semantics

Recall that the semantics basis of the choice calculus is a mapping from decisions to plain variants. In Section 3.4 we argued that the denotational semantics of the choice calculus is compositional since the denotation of an

$$\textsc{W-Shr}\ \frac{\Gamma \vdash e\ \mathit{wf} \qquad \Gamma, v \vdash e'\ \mathit{wf}}{\Gamma \vdash \textbf{share}\ v := e\ \textbf{in}\ e'\ \mathit{wf}} \qquad \textsc{W-Mac}\ \frac{\Gamma \vdash e\ \mathit{wf} \qquad \Gamma, v \vdash e'\ \mathit{wf}}{\Gamma \vdash \textbf{macro}\ v := e\ \textbf{in}\ e'\ \mathit{wf}} \qquad \textsc{W-Var}\ \frac{v \in \Gamma}{\Gamma \vdash v\ \mathit{wf}}$$

Figure 4.8: Extended well-formedness judgment.

$$\begin{aligned}
\lfloor \textbf{share}\ v := e\ \textbf{in}\ e' \rfloor_{D.i} &= \textbf{share}\ v := \lfloor e \rfloor_{D.i}\ \textbf{in}\ \lfloor e' \rfloor_{D.i} \\
\lfloor \textbf{macro}\ v := e\ \textbf{in}\ e' \rfloor_{D.i} &= \textbf{macro}\ v := \lfloor e \rfloor_{D.i}\ \textbf{in}\ \lfloor e' \rfloor_{D.i} \\
\lfloor v \rfloor_{D.i} &= v
\end{aligned}$$

Figure 4.9: Extension of choice elimination.

expression is constructed purely from the expression itself and the denotations of its subexpressions. This is significant since it supports modular language extensions. To add a new language feature, we must only (1) define its syntax as an extension of the choice calculus, (2) extend the choice elimination operation, and (3) define its semantics in terms of the same semantic domain and in a way that satisfies the compositionality property. Then we can reuse all of the existing machinery unchanged.

The previous section (specifically Figure 4.5) shows the extension of the syntax. The extension of the choice elimination operation is straightforward, and given in Figure 4.9. For each of the new constructs, it just propagates the selection to its subexpressions, if any. Most of this section will focus on step (3), showing how the semantics of the **share** and **macro** extensions can be defined compositionally.

Let us first consider the semantics of a **macro** expression. First note that a **macro** expression cannot be simply textually expanded at any place in an expression. Consider the following example, which we'll refer to as *ava*.

$$\begin{array}{l}\textbf{dim } A\langle a,b\rangle \textbf{ in}\\ \textbf{macro } v := A\langle 1,2\rangle \textbf{ in}\\ \textbf{dim } A\langle c,d\rangle \textbf{ in } v\end{array} \qquad (\mathit{ava})$$

Observe that if we textually expand the **macro** expression, the choice that was once bound by the first declaration of dimension A is now bound by the second declaration, as shown below.

$$\begin{array}{l}\textbf{dim } A\langle a,b\rangle \textbf{ in}\\ \textbf{dim } A\langle c,d\rangle \textbf{ in } A\langle 1,2\rangle\end{array}$$

We call this phenomenon *choice capture*. By expanding the **macro** expression, the free choice in the bound expression is captured by a dimension declaration in its scope. This sort of behavior is highly undesirable since it breaks the lexical scoping of dimension names. The situation is analogous to the hygiene issue in other metaprogramming systems [Kohlbecker et al., 1986], where variable capture is known to be error prone.

While this means that we cannot expand **macro** expressions arbitrarily in the middle of expressions, it turns out that we don't need to worry about choice capture when expanding **macro** expressions in the semantic function. To see why this is, recall the semantics of a dimension declaration, repeated below for convenience.

$$[\![\textbf{dim } D\langle t^n\rangle \textbf{ in } e]\!] = \{(D.t_i\,\delta, e') \mid i \in \{1,\ldots,n\}, (\delta, e') \in [\![\lfloor e\rfloor_{D.i}]\!]\}$$

For each tag $D.t_i$ the dimension declares, we simulate the selection of that tag in e, then recursively compute the semantics of the result. Since the **macro** expression in example *ava* is nested within the scope of a dimension declaration, the choice in its bound expression will be eliminated before we compute the semantics of the **macro** expression.

More generally, we can observe that (1) the semantics is defined only for well-formed expressions, (2) a well-formed expression contains no free choices, and (3) choice capture can only occur when there is a free choice in the bound expression. Therefore, we can conclude that the semantic function can expand **macro** expressions without the risk of choice capture. While it is possible for the bound expression to contain locally free choices, as in the example *ava*. These choices are ultimately bound higher in the expression by a dimension declaration, and so will be eliminated before the semantics of the **macro** expression is computed.

Since **share** expressions are resolved at a conceptually later stage than dimensions and choices, there is no risk of choice capture since all variation will have been eliminated by the time the **share** expression is expanded.

Finally, we give the denotational semantics of the **macro** and **share** extensions in Figure 4.10. We use $[e/v]e'$ to represent the variable capture-avoiding substitution of e for every occurrence of v in e'.

The staging distinction between the two constructs can be clearly observed in the semantic function. For **macro** expressions, we expand the expression first by substituting expression e_1 for v in e_2, then compute the semantics of the result. Whereas for **share** expressions, we compute the semantics of the bound expression and the scope first, then substitute each variant e_1' of the bound expression for v in each variant e_2' of the scope. This is similar to the distinction between the call-by-name and call-by-value evaluation strategies for lambda calculus [Pierce, 2002, p. 57].

Note that we must also include a case for variable references, since these must be temporarily preserved in the semantics of the scope of **share** expressions, in order to perform the subsequent substitution. However, the semantics remains undefined for expressions that are not well formed.

4.5 Semantics-Preserving Transformations

In this section we enumerate the semantics-preserving transformations involving the extensions defined in this chapter. Recall from Section 3.5 that we can enumerate these rules by considering every pairwise combination of

$$\begin{aligned}
[\![\textbf{share}\ v := e_1\ \textbf{in}\ e_2]\!] &= \{(\delta_1\,\delta_2,\ [e_1'/v]e_2') \mid (\delta_1, e_1') \in [\![e_1]\!],\ (\delta_2, e_2') \in [\![e_2]\!]\} \\
[\![\textbf{macro}\ v := e_1\ \textbf{in}\ e_2]\!] &= [\![[e_1/v]e_2]\!] \\
[\![v]\!] &= \{(\varepsilon, v)\}
\end{aligned}$$

Figure 4.10: Denotational semantics of share and macro extensions.

constructs, along with constructs for introducing/eliminating each syntactic form. Obviously this results in a multiplicative explosion of equivalence rules as we add more and more constructs to the language, so this section will contain quite a lot of rules!

Since the salient concerns for transforming **share** and **macro** macro expressions are different, we break the presentation into three parts. In Section 4.5.1 we enumerate the rules involving the **share** construct, in Section 4.5.2 we enumerate those involving the **macro** construct, and in Section 4.5.3 we briefly consider the commutation of the two constructs with each other.

4.5.1 Transformations Involving Sharing

Since **share** expressions are expanded after dimensions and choices are resolved, and since the semantics is affected by the ordering of dimension declarations in an expression, transformations involving **share** expressions must be careful not to alter the ordering of dimension declarations. For this reason, many of the rules in this subsection contain premises ensuring that one subexpression or another is dimension free since a dimension-free subexpression can be moved relative to another subexpression without affecting the dimension order. To make these premises nicer, we introduce the predicate $\mathit{free}(e)$ to indicate that expression e is dimension free, otherwise written $BD(e) = \varnothing$.

Shr-Bnd

$$e \equiv \textbf{share}\ v\ := e\ \textbf{in}\ v$$

Shr-Scp

$$\frac{v \notin FV(e) \qquad \mathit{free}(e')}{e \equiv \textbf{share}\ v\ := e'\ \textbf{in}\ e}$$

Figure 4.11: Sharing introduction/elimination rules.

Chc-Shr-Bnd

$$D\langle(\textbf{share}\ v\ := e_i\ \textbf{in}\ e)^{i:1..n}\rangle \equiv \textbf{share}\ v\ := D\langle e^n\rangle\ \textbf{in}\ e$$

Chc-Shr-Scp

$$D\langle(\textbf{share}\ v\ := e\ \textbf{in}\ e_i)^{i:1..n}\rangle \equiv \textbf{share}\ v\ := e\ \textbf{in}\ D\langle e^n\rangle$$

Figure 4.12: Choice-sharing commutation rules.

We begin in Figure 4.11 with rules for introducing or eliminating **share** expressions when applied left to right (LR) or right to left (RL), respectively. The Shr-Bnd rule applied LR can be used to name an expression so that it can be reused later. The Shr-Scp rule applied RL can be used to eliminate a shared expression that is never used. Note, however, that we cannot eliminate the bound expression if it contains a dimension declaration, since this would alter the domain (but not the range) of the denotation of the expression. However, such a transformation *would* be variant preserving (see Section 3.6).

In Figure 4.12 we consider the extension of the choice commutation rules from Figure 3.11 to commute choices with **share** constructs. These rules are significant since applied LR they support the transformation of a sharing-extended choice calculus expression into CNF (see Section 3.5.3). The Chc-Shr-Bnd rule commutes a choice with the bound expression of a **share** expression, while Chc-Shr-Scp commutes a choice with the scope. Note that both rules require that every alternative of the choice on the LHS of the rule

Dim-Shr-Bnd

$$\frac{D \notin FD(e')}{\textbf{dim } D\langle t^n\rangle \textbf{ in } (\textbf{share } v := e \textbf{ in } e') \equiv \textbf{share } v := (\textbf{dim } D\langle t^n\rangle \textbf{ in } e) \textbf{ in } e'}$$

Dim-Shr-Scp

$$\frac{D \notin FD(e) \qquad \mathit{free}(e)}{\textbf{dim } D\langle t^n\rangle \textbf{ in } (\textbf{share } v := e \textbf{ in } e') \equiv \textbf{share } v := e \textbf{ in } (\textbf{dim } D\langle t^n\rangle \textbf{ in } e')}$$

Figure 4.13: Dimension-sharing commutation rules.

contain a **share** expression that differs only in the alternatives of the choice on the RHS. This is not such an onerous restriction since we can use the introduction/elimination rules in Figure 4.11 to either introduce new **share** expressions to set up a LR application of a Chc-Shr rule, or to eliminate unwanted **share** expressions after a RL application of a Chc-Shr rule.

In Figure 4.13 we extend the dimension commutation rules from Figure 3.13 to commute dimensions and **share** expressions. Once again, we can factor dimension declarations in either the bound expression or the scope of a **share** expression. Considering the rules applied RL, we can lift a dimension out of the bound expression (Dim-Shr-Bnd) only if doing so would not capture a choice in the scope. Likewise, we can lift a dimension out of the scope (Dim-Shr-Scp) only if it would not capture a choice in the bound expression. In rule Dim-Shr-Scp we must also ensure that lifting the dimension declaration out of the scope will not alter the order of dimension declarations by requiring the bound expression to be dimension free.

Finally, in Figure 4.14 we consider the commutation of sharing with object structures and with other **share** expressions. The rule for factoring a **share** expression out of a bound expression (Shr-Shr-Bnd applied LR) is the simplest since it does not affect the ordering of the subexpressions; we must only

Shr-Shr-Bnd

$$\frac{w \notin FV(e'')}{\textbf{share}\ v\ := (\textbf{share}\ w\ := e\ \textbf{in}\ e')\ \textbf{in}\ e'' \equiv \textbf{share}\ w\ := e\ \textbf{in}\ \textbf{share}\ v\ := e'\ \textbf{in}\ e''}$$

Shr-Shr-Scp

$$\frac{v \neq w \qquad v \notin FV(e') \qquad w \notin FV(e) \qquad \mathit{free}(e) \vee \mathit{free}(e')}{\textbf{share}\ v\ := e\ \textbf{in}\ \textbf{share}\ w\ := e'\ \textbf{in}\ e'' \equiv \textbf{share}\ w\ := e'\ \textbf{in}\ \textbf{share}\ v\ := e\ \textbf{in}\ e''}$$

Shr-Obj

$$\frac{v \notin \cup_{j \neq i} FV(e_j) \qquad \mathit{free}(e) \vee (\mathit{free}(e_1) \wedge \ldots \wedge \mathit{free}(e_{i-1}))}{\textbf{share}\ v\ := e\ \textbf{in}\ a{\prec}e^n[i : e']{\succ} \equiv a{\prec}e^n[i : \textbf{share}\ v\ := e\ \textbf{in}\ e']{\succ}}$$

Figure 4.14: Remaining sharing commutation rules.

ensure that doing so does not capture a free w variable in e''. When factoring a **share** expression out of the scope of another **share** expression, however, we must ensure that the order of dimensions is not altered. Since the expressions e and e' swap relative positions in the expression, this transformation is semantics preserving only if either or both of the expressions e and e' are dimension free. For the same reason, when factoring a **share** expression out of an object structure, we require that either the bound expression e is dimension free, or that every subexpression of the structure occurring before the **share** expression is dimension free. For both the Shr-Shr-Scp and Shr-Obj rules, the remaining premises ensure that the transformation does not result in any variables escaping or becoming captured.

$$\textsc{Mac-Bnd}\quad e \equiv \textbf{macro}\ v := e\ \textbf{in}\ v \qquad\qquad \textsc{Mac-Scp}\quad \frac{v \notin FV(e)}{e \equiv \textbf{macro}\ v := e'\ \textbf{in}\ e}$$

Figure 4.15: Macro introduction/elimination rules.

4.5.2 Transformations Involving Macros

In Figure 4.15, we begin as before with rules for introducing or eliminating **macro** expressions, depending on whether they are applied LR or RL, respectively. These rules are almost identical to the **share** introduction/elimination rules in Figure 4.11, except that in Mac-Scp we do not require that e' be dimension free. The reason is that since **macro** expressions are expanded before dimension elimination, if e does not reference v, then any dimensions in e' will be eliminated by the **macro** expansion before they have an effect on the semantics.

More generally, in contrast to **share** expressions, when commuting **macro** expressions we can ignore dimension ordering entirely. This is because the order that dimensions are eliminated is determined by the location(s) that a macro is *referenced* rather than the location it is *defined*. This makes commuting **macro** expressions somewhat easier than **share** expressions.

Figure 4.16 shows the commutation of macros with both choices and dimension declarations. As with **share** expressions, we require that every alternative of a choice have a similar **macro** expression in order to commute macros with choices. These can be introduced by the rules Figure 4.15 to set up a LR application of the Chc-Mac-Bnd rules, or eliminated by the same rules to clean up after a RL application of the Chc-Mac-Bnd rules.

Observe that we can freely commute dimension declarations in the scope of a **macro** expression (Dim-Mac-Scp). However, there is no rule Dim-Mac-Bnd since a dimension declaration bound in a **macro** expression can be potentially

Chc-Mac-Bnd

$$D\langle(\textbf{macro}\ v := e_i\ \textbf{in}\ e)^{i:1..n}\rangle \equiv \textbf{macro}\ v := D\langle e^n\rangle\ \textbf{in}\ e$$

Chc-Mac-Scp

$$D\langle(\textbf{macro}\ v := e\ \textbf{in}\ e_i)^{i:1..n}\rangle \equiv \textbf{macro}\ v := e\ \textbf{in}\ D\langle e^n\rangle$$

Dim-Mac-Scp

$$\textbf{macro}\ v := e\ \textbf{in}\ (\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e') \equiv \textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ (\textbf{macro}\ v := e\ \textbf{in}\ e')$$

Figure 4.16: Commuting macros with choices and dimensions.

duplicated many times. In the special cases where v is referenced never or once in the scope, we can apply a sequence of other rules to remove the dimension declaration. For example, if v is referenced just once, we can push the macro definition down to the reference of v, then eliminate it by a RL application of Mac-Bnd.

Figure 4.17 considers the commutation of macros with other **macro** expressions and with object structures. These rules are very similar to the corresponding rules for **share** expressions, presented in Figure 4.14, except that we do not need to preserve the relative orderings of subexpressions that contain dimension declarations. For example, in rule Mac-Mac-Scp, even if e and e' both contain dimension declarations, the semantics is unchanged by the transformation since the macros are expanded before dimensions are eliminated. Therefore, as long as we preserve which bound expression each variable refers to, we can reorder subexpressions freely.

Mac-Mac-Bnd

$$\frac{w \notin FV(e'')}{\textbf{macro}\ v := (\textbf{macro}\ w := e\ \textbf{in}\ e')\ \textbf{in}\ e'' \equiv \textbf{macro}\ w := e\ \textbf{in}\ \textbf{macro}\ v := e'\ \textbf{in}\ e''}$$

Mac-Mac-Scp

$$\frac{v \neq w \qquad v \notin FV(e') \qquad w \notin FV(e)}{\textbf{macro}\ v := e\ \textbf{in}\ \textbf{macro}\ w := e'\ \textbf{in}\ e'' \equiv \textbf{macro}\ w := e'\ \textbf{in}\ \textbf{macro}\ v := e\ \textbf{in}\ e''}$$

Mac-Obj

$$\textbf{macro}\ v := e\ \textbf{in}\ a\prec e^n \succ\ \equiv a\prec(\textbf{macro}\ v := e\ \textbf{in}\ e_i)^{i:1..n}\succ$$

Figure 4.17: Remaining macro commutation rules.

4.5.3 Commuting the Two Extensions

Finally, we consider how the two extensions can be commuted with each other. Figure 4.18 enumerates all of the possible commutations of **share** and **macro** expressions. Although the rules are again similar to the commutation rules in Figure 4.14 and Figure 4.17, reasoning about the interaction of these two constructs is complicated by the fact that they are expanded in different stages relative to dimension declaration. However, we can exploit these same staging constraints to simplify the situation. The trick is to simulate the expansion of the macro definitions first, then compare the semantics on each side of the equivalence.

Since most of the rules involve only a single **share** expression, the relative ordering of dimension declarations after macro expansion is preserved without the need for additional premises. The exception is the Mac-Shr-Bnd rule. When applied LR, this rule can duplicate the **share** expression bound to v, resulting in potentially many copies of e. Therefore we must ensure that e is dimension free. Note that although e' is also duplicated by the **macro**

Shr-Mac-Bnd

$$\frac{w \notin FV(e'')}{\textbf{share } v := (\textbf{macro } w := e \textbf{ in } e') \textbf{ in } e'' \equiv \textbf{macro } w := e \textbf{ in share } v := e' \textbf{ in } e''}$$

Shr-Mac-Scp

$$\frac{v \neq w \qquad v \notin FV(e') \qquad w \notin FV(e)}{\textbf{share } v := e \textbf{ in macro } w := e' \textbf{ in } e'' \equiv \textbf{macro } w := e' \textbf{ in share } v := e \textbf{ in } e''}$$

Mac-Shr-Bnd

$$\frac{w \notin FV(e'') \qquad \mathit{free}(e)}{\textbf{macro } v := (\textbf{share } w := e \textbf{ in } e') \textbf{ in } e'' \equiv \textbf{share } w := e \textbf{ in macro } v := e' \textbf{ in } e''}$$

Mac-Shr-Scp

$$\frac{v \neq w \qquad v \notin FV(e') \qquad w \notin FV(e)}{\textbf{macro } v := e \textbf{ in share } w := e' \textbf{ in } e'' \equiv \textbf{share } w := e' \textbf{ in macro } v := e \textbf{ in } e''}$$

Figure 4.18: Commuting sharing and macros.

expansion, it is duplicated on the RHS of the equivalence too, so it may contain dimension declarations.

Chapter 5 – Internalized Selection

In Section 2.2 we have described the three major roles that metalanguages serve in the process of managing variation in software and other artifacts. In terms of the choice calculus we can describe these roles as follows.

1. Metalanguages are used to *organize* or *model* the variation space by defining the set of valid configurations. In the choice calculus this is supported by separating the decision space into dimensions of variation. Relationships between these dimensions (such as dependencies) can be expressed by the nesting of dimensions within choices.

2. Metalanguages are used to *implement* the variation in the artifact. In the choice calculus, variability is expressed by choices that locally capture variation points between similar artifacts.

3. Metalanguages are used to *configure* an individual variant from a variational artifact. In the choice calculus this is achieved by the process of tag selection, which can be used to eliminate dimensions of variation until a particular plain variant is achieved.

The choice calculus as defined in Chapter 3 supports variation organization and implementation as first-class concepts in the language itself but configuration is defined only externally through meta-theoretic operations. This asymmetry is apparent in the syntax of the choice calculus, which provides constructs for dimension declarations and choices, but not for selection.

In this chapter we extend the choice calculus with a syntactic form for selection. This construct provides first-class support for configuration in the choice calculus, resolving the asymmetry described above. We briefly introduce the notation and high-level expectations of this extension in Section 5.1, and further motivate it in Section 5.2 by describing some specific benefits and use cases.

As with sharing, the addition of a selection construct poses many challenging design questions. In Section 5.3 we present several possible interpretations of selection, analyze their trade-offs, and explore how selection interacts with other choice calculus constructs. In Section 5.4 we pick a particular interpretation and formally define the syntax and denotational semantics of the extension.

The addition of syntactic selection makes the well-formedness property (see Section 3.3.3 and Section 4.3) much more interesting. In Section 5.5 we expand this into a type system that associates a *configuration type* with each well-formed choice calculus expression. A configuration type encodes the structure of the decision space of an expression, revealing the sequences of selections that must be made in order to resolve an expression into one of the plain variants it encodes.

Finally, in Section 5.6 we enumerate the semantics-preserving transformations involving the new selection construct, just as we did for the reuse constructs introduced in the previous chapter.

5.1 Representing Selection in the Choice Calculus

Although the formal definition of the language extension is deferred until later in the chapter, in this section we briefly introduce the notation of internalized selection and describe the intuition behind its expected behavior. This intuition is the basis for the discussion in Section 5.2 and Section 5.3.

We use the syntax **select** $D.t$ **from** e to represent the selection of tag t from dimension D in the *target* expression e. The precise meaning of such an expression will be explored in the next section and finalized in Section 5.4. Intuitively, however, evaluating a **select** expression should correspond to the tag selection operation that is part of the denotational semantics defined in

Section 3.4. For example, consider the following simple expression e_{ab} that declares a single dimension A.

$$\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle \mathtt{1},\mathtt{2}\rangle \qquad (e_{ab})$$

The expected semantics of selecting $A.a$ in e_{ab} is clear.

$$[\![\mathbf{select}\ A.a\ \mathbf{from}\ e_{ab}]\!] = \{(\varepsilon,\mathtt{1})\}$$

When the **select** expression is evaluated, the dimension A is eliminated from e_{ab} and all bound choices are replaced by their first alternative, which corresponds to the tag a. Since the only declared dimension in e_{ab} was eliminated by a **select** expression, there are no selections left to make, so the semantics is just a mapping from the empty decision to the variant `1`.

We call a dimension D *exposed* in an expression if a selection in D will have some effect. For example, in the expression e_{ab}, the dimension A is exposed. Dimension A is also exposed in $\mathtt{1+}e_{ab}$ but not in **select** $A.b$ **from** $\mathtt{1+}e_{ab}$ since the selection eliminates A and any subsequent selection in A would either be an error or have no effect (see Section 5.3.1). Additionally, both dimensions A and B are exposed in the expression $\mathbf{dim}\ B\langle c,d\rangle\ \mathbf{in}\ e_{ab}$ since we can make a selection in either or both dimensions.

5.2 Motivation

There are many practical and theoretical motivations for representing selection explicitly within the choice calculus itself. From a practical perspective, one of the goals of the choice calculus is to model existing variation management systems, and many such systems provide a way to configure or partially configure variants within the system itself. One simple example of this is CPP's `#define` directive, which can be used to change or set the value of a CPP macro during the preprocessing phase. More generally, there is research on *nested software product* lines, where product lines occur as reusable components within a larger product line that must be individually configured at each use [Krueger, 2006, Rosenmüller et al., 2008, Rosenmüller and Siegmund, 2010]. In order for the choice calculus to model such systems, it must provide a language-level mechanism for configuring variational components.

From a theoretical perspective, internalizing selection greatly increases our ability to study the nature of configuration, for example, by identifying semantics-preserving transformations, defining type systems, and exploring alternative semantics. Essentially, by promoting selection from an external operation to a language feature, we can examine its effects and interactions with other features more directly.

In the rest of this section we present a few more arguments and use cases that demonstrate the utility of the **select** construct. In Section 5.2.1 we show how the selection construct can be used to support tag and dimension renaming. In Section 5.2.2 we describe how it can improve the modularity of choice calculus expressions. And in Section 5.2.3 we describe how language-level support for selection supports the definition of an operational semantics for the choice calculus.

5.2.1 Dimension and Tag Renaming

The choice calculus's **dim** construct provides a way to declare and scope new dimensions of variation, and to associate tag names with each alternative in a dimension. This provides quite a bit of flexibility in managing the namespace of dimensions and tags. We can have multiple independent dimensions with the same name in the same choice calculus expression, and we can have tags with the same name in different dimensions.

However, once a dimension or tag has been declared, the only way to change its name is to edit the choice calculus expression directly. This has many drawbacks. In the case of renaming dimensions, the same edit must be made at many places since we must also rename every choice bound by that dimension. Although such a transformation could be automated, the fact is that renaming a dimension is quite an invasive operation potentially affecting a lot of code. This invasiveness is a much bigger problem in a collaborative setting since we may not even have access to the original expression to perform the renaming operation.

More importantly, the lack of renaming fundamentally limits the potential for reusing independently developed code that might inadvertently use the same dimension names. Since selections in like-named dimensions are resolved in a fixed, top-to-bottom, left-to-right order (see Section 3.4), clashing dimension names can make it impossible to partially configure reused code in the desired way. This issue is discussed in more depth in the next subsection.

As a solution, the **select** construct can be used to rename and reorder dimension and tag names in a non-invasive way. For example, if expression e exposes a dimension A with tags a and b, but we would rather expose a dimension B with tags c and d, we can implement the renaming as follows.

$$\begin{array}{l} \textbf{dim}\ B\langle c, d\rangle\ \textbf{in} \\ B\langle \textbf{select}\ A.a\ \textbf{from}\ e, \textbf{select}\ A.b\ \textbf{from}\ e\rangle \end{array}$$

This construction is not invasive since the implementation of e is unchanged. If we also want to avoid duplicating e, we can also make use of the **macro** extension from the previous chapter.

$$\begin{array}{l} \textbf{macro}\ v := e\ \textbf{in} \\ \textbf{dim}\ B\langle c, d\rangle\ \textbf{in} \\ B\langle \textbf{select}\ A.a\ \textbf{from}\ v, \textbf{select}\ A.b\ \textbf{from}\ v\rangle \end{array}$$

Note that this assumes, however, that macros are expanded at an earlier stage than selections are resolved. This design decision is discussed in Section 5.3.5.

In addition to renaming dimensions and tags, the strategy applied above can be used to *partially* reduce the variability in a dimension. For example, if an expression e exposes a dimension A with tags a, b, and c, then we can wrap e in a declaration of A with only tags a and b, using the strategy above to effectively eliminate the alternatives corresponding to $A.c$.

5.2.2 Modularity of Variational Components

In the previous subsection we saw how the **select** construct supports the reuse of variational components by providing a mechanism for externally renaming the exposed dimensions of a component. More generally, a syntactic form for selection allows us to precisely control how the variability of a component affects the overall variability of the system.

In terms of the choice calculus, a variational component is just a choice calculus expression whose variability is expressed by its exposed dimensions. A component may be reused in many places by a **macro** expression and components may be nested within other components. The overall variability of the system corresponds to the exposed dimensions of the whole choice calculus expression.[1]

We can distinguish three ways that an exposed dimension in a component can contribute to the overall variability of the system (or to the larger component of which it is a part).

- An exposed dimension can *directly contribute* to the overall variability of the system. That is, the dimension in the component is exposed to clients of the overall system. This is the default behavior of the choice calculus without syntactic support for selection.

- An exposed dimension can be *locally resolved* by a **select** expression. That is, at the point that the component is reused, it is (partially or fully) configured as an implementation detail of the system. The fact that this variability ever existed is hidden from clients of the overall system, ensuring proper information hiding.

- An exposed dimension can *indirectly contribute* to the overall variability of the system. That is, the variability in the component is explicitly mapped to one or more dimensions of variation exposed by the overall

[1] In Section 5.5 we will develop a more sophisticated representation of the variability of an expression in the form of *configuration types*.

system. This can range from a simple renaming or partial configuration, as illustrated in Section 5.2.1, to the construction of complex dependencies with several dimensions, which can be constructed in a similar manner by nested **dim** and **select** expressions.

In this way, the **select** construct allows us to precisely control the *configuration interface* that an expression presents to clients—that is, the set of decisions that clients can make about an expression—even if we don't have direct access to each of the variational components used in the construction of that expression. We believe that this kind of control is crucial to support code reuse and for scaling applications of the choice calculus to large systems.

5.2.3 Operational Semantics

Finally, a syntactic form for selection provides a natural path toward defining an operational semantics for the choice calculus in terms of syntactic transformations on choice calculus expressions. For example, if we require that all choice calculus expressions be *fully configured* in the sense that they expose no dimensions of variation, then a structured operational semantics [Plotkin, 1981] might be defined by rules that:

1. Incrementally move **select** constructs closer to the **dim** declarations that they resolve.

2. Eliminate **select**-**dim** sequences in the same dimension.

Such a reduction would eventually yield the single plain variant that the fully configured choice calculus expression represents. We expect the elimination of **select**-**dim** sequences to satisfy an equivalence like the following, where t_i represents the ith tag in the sequence t^n.

$$\textbf{select}\ D.t_i\ \textbf{from}\ (\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e) \equiv \lfloor e \rfloor_{D.i}$$

Of course, we would also need rules for expanding **macro** and **share** expressions, and congruence rules for other pairs of syntactic forms.

Even though we prefer a denotational semantics for the choice calculus since it is simple, compositional, and extensible, an operational semantics has certain advantages as well. For example, operational semantics correspond more directly to implementations, which make it easier to reuse theoretical results in the development of tools. Additionally, a structural operational semantics can better reveal where an expression "went wrong" in cases where the semantics is undefined by providing a partial reduction sequence.

We do not define an operational semantics here, but by internalizing the operation for eliminating dimensions of variation from choice calculus expressions, the **select** construct makes such a definition possible. This is something we plan to pursue in future work.

5.3 Design Questions

In many scenarios, the intended meaning of a selection is not obvious. In this section, we collect a number of challenges for a formal definition of selection. We discuss the merits and drawbacks of each potential resolution, along with the often subtle interactions between these design decisions.

5.3.1 Undeclared Dimensions

The first question is what to do when the dimension of a selected qualified tag is not declared in the target of the selection. This is illustrated by the following example.

select $D.t$ **from** `1 + 2`

The meaning of selection in an undeclared dimension can be defined in at least two ways.

1. The selection is considered ill-formed, in which case the semantics is undefined and an error is reported.
2. The selection is idempotent, in which case our example is equivalent to the expression `1 + 2`.

On the one hand, the purpose of selection is to eliminate dimensions, so a selection that does not do this can be considered anomalous and should perhaps be identified as such. However, the idempotent behavior permits a larger set of well-defined expressions and more semantics-preserving transformations, as we will see later in this section. For this reason, we choose that selections in undeclared dimensions be idempotent.

5.3.2 Multiple Matching Dimensions

The next question is what to do when there are multiple matching dimension declarations in parallel. This situation is encountered when dimension declarations are nested in different siblings of an object structure, as in the following example.

$$\begin{array}{l}\textbf{select}\ D.a\ \textbf{from}\\ (\textbf{dim}\ D\langle a,b\rangle\ \textbf{in}\ D\langle 1,2\rangle)\ +\ (\textbf{dim}\ D\langle a,c\rangle\ \textbf{in}\ D\langle 3,4\rangle)\end{array}$$

There are at least four possible resolutions:

1. We allow a selection to be applied only directly to the declaration of the dimension that it eliminates. Although restrictive, this prevents the multiple matching dimension issue (and many other ambiguities) from arising. In this case, the example is ill-formed and an error is reported.
2. A selection that matches multiple dimension declarations in parallel is considered ambiguous. In this case, the example is ambiguous and an error is reported.

3. The leftmost matching dimension declaration is resolved. In this case, the example is equivalent to `1 +` $(\textbf{dim}\ D\langle a, c\rangle\ \textbf{in}\ D\langle 3, 4\rangle)$. This interpretation corresponds most closely to the external selection operation that is the basis of the denotational semantics defined in Section 3.4. We call this behavior of eliminating at most one dimension with one selection *modest selection*.

4. All matching dimension declarations are resolved. In this case, the example is equivalent to the plain expression `1 + 3`. We call this behavior of removing all matching, parallel dimension declarations with one selection *greedy selection*.

Before we discard the first resolution as being too restrictive, it is worth noting that it has some nice properties and parallels in other languages. For example, requiring a **select** to be applied directly to the **dim** declaration it eliminates is not so different from the lambda calculus, where lambda abstractions are reduced only when applied directly to an argument. This may help to enforce modularity since only a dimension declaration at the root of an expression can be referenced and eliminated. In our previous work on this extension we have explored a simple module system that relies on and enforces this constraint [Erwig et al., 2013a].

However, the first resolution also leads to an extension that has low orthogonality since the **select** construct cannot be combined freely with other syntactic forms. It also leads to a situation where we cannot externally configure variational components, which was one of the motivations for the selection extension, described in Section 5.2.2. For these reasons we reject this resolution to the multiple-matching-dimension design question.

We also reject the second resolution since it permits fewer valid expressions than either modest or greedy selection, and it does not support the modularity of variational components that declare multiple dimensions with the same name.

This leaves a choice between the modest and greedy forms of selection. A simple observation is that both interpretations are equivalent when the expression is dimension linear. Recall from Section 3.5.3 that an expression is dimension linear if all of its declared dimensions have pairwise different names. In the absence of dimension linearity, however, there are some interesting trade-offs between the two approaches.

The first observation is that the greedy interpretation is more useful for pushing selections *down* an expression, while modest selection is more useful for lifting selections *up* an expression. To illustrate, observe that the greedy interpretation suggests the following equivalence relation for commuting selections and object structures.

$$\textbf{select}\ D.t\ \textbf{from}\ a\prec e^n \succ\ \equiv\ a\prec(\textbf{select}\ D.t\ \textbf{from}\ e_i)^{i:1..n}\succ$$

Applied left-to-right, this relation can be used to push selection operations down to the declarations of the dimensions that they affect, leading to a simple, purely syntactic account of selection. Note that there is an interaction here with the undeclared dimension design question from Section 5.3.1 since the relation above only holds if we assume that the selection of an undeclared dimension is idempotent.

With the modest interpretation, a selection can only be pushed down an object structure by examining each of the subexpressions to determine which subexpression declares the dimension that will be matched. Worse, in the presence of **macro** expressions, the dimension that is matched may depend on the context surrounding the **select** expression since a macro may be expanded in a way that introduces new dimension declarations (see Section 5.3.5 and Section 5.6 for more on this issue). In other words, pushing a selection down an expression is not a local syntactic transformation [Felleisen, 1991]. This makes greedy selection a better choice for a structural operational semantics, as described in Section 5.2.3.

To see how the modest approach better supports lifting selections up an expression, consider the following expression with a selection in the *i*th subexpression of an object structure.

$$a{\prec}e^n[i : \textbf{select } D.t \textbf{ from } e_i]{\succ}$$

With the greedy approach, the selection can only be factored out of the object structure if none of the other subexpressions expose a dimension named D, otherwise this dimension would be captured by the lifted selection. In contrast, with the modest approach, we can factor out the selection by prioritizing e_i over the preceding subexpressions $e_1, \ldots, e_{i-1}$ by introducing a **share** construct, as shown below.

$$\begin{array}{l}\textbf{select } D.t \textbf{ from}\\ \textbf{share } v := e_i \textbf{ in } a{\prec}e^n[i : v]{\succ}\end{array}$$

The ease with which selections can be lifted reflects the fact that modest selection more closely implements the external selection operation used in the denotational semantics. Under this interpretation, the resolution of a **select** expression at the top of an expression is equivalent to external selection.

Modest selection also better supports the modularity of variational components. If a variational component declares two independent dimensions with the same name, greedy selection forces them to be synchronized while modest selection allows them to be configured separately.

The choice of greedy or modest selection interacts with many of the other design questions described in this section, so we will refer back to both of these interpretations of selection. Ultimately, however, we choose the modest form of selection since it corresponds most directly to the external selection operation we have assumed so far and since it supports the separate configuration of independent dimensions in variational components.

5.3.3 Undeclared Tags

The next design question is what to do when the dimension referred to by a selection is declared in the target, but the declaration does not contain the selected tag. This problem is illustrated in the following example.

$$\begin{array}{l}\textbf{select}\ D.a\ \textbf{from}\\ \textbf{dim}\ D\langle b,c\rangle\ \textbf{in}\ D\langle 1,2\rangle\end{array}$$

Taking into account that we consider selections in undeclared dimensions to be idempotent, there are two ways to resolve selection with an undeclared tag name:

1. The problem could be handled in the same way as selection with an undeclared dimension. In this case, the selection of an undeclared tag is idempotent and the example is equivalent to $\textbf{dim}\ D\langle b,c\rangle\ \textbf{in}\ D\langle 1,2\rangle$.

2. The selection of an undeclared tag in a declared dimension could be considered ill-formed, in which case an error is reported.

The argument for the first view is a simple appeal to consistency. The argument for the second is more subtle, and is best viewed through the interaction of this design decision with the treatment of parallel matching dimension declarations, discussed in the previous subsection. Consider the following example.

$$(\textbf{dim}\ D\langle b,c\rangle\ \textbf{in}\ D\langle 1,2\rangle)\ +\ (\textbf{dim}\ D\langle a,b\rangle\ \textbf{in}\ D\langle 3,4\rangle)$$

If we assume the first resolution, then this example exhibits a strange asymmetry with both modest and greedy selection:

- With modest selection, observe that from the top of this expression we can select either $D.b$ or $D.c$ from the left dimension, or $D.a$ from the right dimension, but not $D.b$ from the right dimension (because it is shadowed by the tag b in the first declaration of D).

- With greedy selection, if we choose $D.b$, the two dimensions will be synchronized as expected, but if we first choose $D.a$ or $D.c$, then only one of the dimensions will be eliminated and we can unilaterally choose $D.b$ in the remaining dimension.

The second resolution resolves the asymmetry for the case of modest selection. By making the selection of $D.a$ an error, we are forced to select tags from the two dimensions in order, left to right. However, for greedy selection the second resolution seems overly restrictive since it makes the selection of either $D.a$ or $D.c$ an error, effectively forcing all parallel dimension declarations with the same name to declare the same tags. There is no clear resolution of the asymmetry for greedy selection.

Fortunately, since we have already chosen modest selection, we will also choose the second resolution here since it does not exhibit the asymmetry of the more consistent first resolution. Therefore, we consider the selection of an undeclared tag in a declared dimension to be ill-formed.

5.3.4 Dependent Dimensions

The next design question is how to treat selection from a dimension that is nested within a choice in a different dimension. Recall from Section 3.3.2 that the nested dimension is said to be *dependent* on the selection of the corresponding tag(s) in the outer dimension. For example, in the following expression the dimension B is dependent on the selection of either tag a_1 or tag a_2 in dimension A.

$$\begin{array}{l}\textbf{select}\ B.b_1\ \textbf{from}\\ \textbf{dim}\ A\langle a_1,a_2,a_3\rangle\ \textbf{in}\\ A\langle \textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle 1,2\rangle, \textbf{dim}\ B\langle b_1,b_2\rangle\ \textbf{in}\ B\langle 3,4\rangle, 5\rangle\end{array}$$

There are several possible interpretations of this scenario.

1. We require that outer dimensions be resolved before dependent dimensions. If a **select** expression refers to a dependent dimension there are two possible interpretations.
   (a) Since dependent dimensions cannot be selected, the situation is analogous to selecting an undeclared dimension. By the design decision in Section 5.3.1, the selection is idempotent.
   (b) Selection in a dependent dimension is an error. In this case, the example is ill-formed and an error is reported.
2. The matching dimension declaration in the leftmost alternative is resolved, preserving the other alternatives of the choice. The example is equivalent to $\textbf{dim}\ A\langle a_1, a_2, a_3\rangle\ \textbf{in}\ A\langle 1, \textbf{dim}\ B\langle b_1, b_2\rangle\ \textbf{in}\ B\langle 3, 4\rangle, 5\rangle$.
3. The selection is mapped across the choice so that matching dimension declarations in every alternative are resolved. For alternatives that do not expose a matching dimension, there are two possibilities.
   (a) Alternatives that do not expose the dependent dimension are preserved. The example is equivalent to $\textbf{dim}\ A\langle a_1, a_2, a_3\rangle\ \textbf{in}\ A\langle 1, 3, 5\rangle$.
   (b) Alternatives that do not expose the dependent dimension are removed. The example is equivalent to $\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle 1, 3\rangle$.

The main argument for the first resolution is that it is most consistent with the current external definition of tag selection. With external selection, dimensions are strictly ordered so that dependent dimensions are always selected after their dependencies are resolved. This ordering constraint seems quite restrictive for internalized selection, however, and we have already loosened it by allowing independent dimensions to be selected in any order. Therefore a consistency argument might also be made for *allowing* dependent dimensions to be selected.

Allowing selection in dependent dimensions also better supports the reuse of variational components since it supports a form of *conditional configuration*.

For example, suppose we are developing an application with variants for Windows and Mac. A variational component e exposes a top-level dimension $\textbf{dim}\ \mathit{OS}\langle \mathit{Windows}, \mathit{Unix}\rangle$, and a dimension $\textbf{dim}\ \mathit{Flavor}\langle \mathit{Linux}, \mathit{Mac}\rangle$ that is dependent on the selection of $\mathit{OS}.\mathit{Unix}$. Now we can conditionally select $\mathit{Flavor}.\mathit{Mac}$ if $\mathit{OS}.\mathit{Unix}$ is selected with the expression $\textbf{select}\ \mathit{Flavor}.\mathit{Mac}\ \textbf{from}\ e$. This restricts e to the variants corresponding to Widows and Mac, removing the variants corresponding to the Linux flavor of Unix.

Since conditional selections provide a practical use case for admitting a larger class of expressions, and since we have already lifted some of the ordering constraints imposed by external selection, we choose to allow selections in dependent dimensions.

The second interpretation of selection in a dependent dimension (eliminating the leftmost matching alternative) is at first appealing since it seems consistent with the modest semantics we chose as a solution to the multiple parallel dimensions design question in Section 5.3.2. However, consider the following pair of expressions, which are equivalent by the Chc-Dim transformation law presented in Section 3.5.

$$\begin{array}{lcl}
\textbf{dim}\ C\langle c_1, c_2\rangle\ \textbf{in} & & \textbf{dim}\ C\langle c_1, c_2\rangle\ \textbf{in} \\
\textbf{dim}\ D\langle d_1, d_2\rangle\ \textbf{in} & \equiv & C\langle \textbf{dim}\ D\langle d_1, d_2\rangle\ \textbf{in}\ D\langle 1, 2\rangle, \\
C\langle D\langle 1, 2\rangle, D\langle 3, 4\rangle\rangle & & \quad \textbf{dim}\ D\langle d_1, d_2\rangle\ \textbf{in}\ D\langle 3, 4\rangle\rangle
\end{array}$$

Under the second interpretation, if we apply $\textbf{select}\ D.d_1$ to each each expression, we will get different results despite the fact that the two expressions are semantically equivalent. This is bad since it means the Chc-Dim rule is no longer semantics-preserving in the presence of the **select** extension.

Therefore it seems that although we *fold* selections across object structures until they are consumed, we must *map* selections across choices. This leads us to choose the third interpretation of selection in a dependent dimension, where we eliminate matching dimensions in all alternatives. However, we must still decide what to do with alternatives that do contain a matching dimension declaration. The two possibilities are listed in interpretations 3(a) and 3(b).

The argument for interpretation 3(b) is that selection in a dependent dimension can be viewed as an implicit selection of the tags that dimension is dependent on. As an example, consider the following excerpt from a variational list of errands.

$\textbf{select}\ \mathit{BuyPie.yes}\ \textbf{from}$
$\textbf{dim}\ \mathit{VisitBakery}\langle \mathit{yes}, \mathit{no}\rangle\ \textbf{in}$
$\mathit{VisitBakery}\langle \textbf{dim}\ \mathit{BuyPie}\langle \mathit{yes}, \mathit{no}\rangle\ \textbf{in}\ \mathit{BuyPie}\langle \texttt{buy pie}, \texttt{say hi}\rangle, \ldots\rangle$

Since we have decided to buy the pie (based on the selection of *BuyPie.yes*), then we must necessarily visit the bakery. So we can consider the above expression equivalent to the following, in which we are now forced to select *yes* in the *VisitBakery* dimension.

$\textbf{dim}\ \mathit{VisitBakery}\langle \mathit{yes}\rangle\ \textbf{in}\ \mathit{VisitBakery}\langle \texttt{buy pie}\rangle$

Note that we do not directly select *VisitBakery.yes* but only eliminate the tag *VisitBakery.no* and its corresponding alternatives. The reason is that a dimension can be dependent on more than one tag in the same dimension, as in the example with dimensions A and B at the beginning of this subsection. Filtering out unmatched alternatives will generalize to this cases, while implicitly selecting dependencies will not. However, we might complement the filtering step with a subsequent rule that eliminates dimensions with only a single tag, as supported by Theorem 3.6.6, so that the above example is equivalent to simply the plain expression `buy pie`.

The drawback of this interpretation is that it is quite complicated and perhaps better described as a combination of more primitive operations. It also removes the chance to make conditional selections, as described in the *OS* example above.

Therefore, by a process of elimination, we choose interpretation 3(a) where selections in dependent dimensions are mapped across all of the alternatives of a choice but unmatched alternatives are retained.

5.3.5 Staging and Scope of Selection

The final challenge is to determine at which conceptual stage **select** expressions are resolved. This is significant to determine how selection interacts with the reuse extensions presented in the previous chapter. Recall from Section 4.2 that the evaluation of an expression in the fully extended choice calculus consists of three conceptual stages, resolved in order: (1) **macro** expansion, (2) **dim** elimination, and finally (3) **share** expansion. Where does selection fit into this view?

Recall from Section 5.2.2 that one of the motivations for an internalized selection operation is to support the modularity and (partial) configuration of variational components. From this perspective, it only makes sense to resolve **select** expressions *after* **macro** expressions since we may want to configure variational components differently at each point where they are reused. Additionally, in order to have an effect, **select** expressions must be resolved *before* **dim** elimination. This establishes a clear stage between **macro** expansion and **dim** elimination in which **select** resolution must occur.

However, resolving selections after macros leads to a subsequent issue of whether the target of a selection should be determined lexically or dynamically. In other words, does the expansion of macros alter the set of dimensions that can be selected at each point in the program? This issue can be clearly seen in the following example.

$$\begin{array}{l}\textbf{macro}\ v := (\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle 1, 2\rangle)\ \textbf{in}\\ \textbf{select}\ A.a_1\ \textbf{from}\ v\end{array}$$

There are two possible ways to interpret this expression.

1. The target of a selection is determined lexically. In this case, even though the dimension declaration is expanded into the scope of the **select** expression, the selection has no effect and the example is equivalent to **dim** $A\langle a_1, a_2\rangle$ **in** $A\langle 1, 2\rangle$.

2. The target of a selection is determined dynamically. In this case, the macro expansion causes the declaration of A to be captured by the **select** expression. The dimension A is eliminated and the example is equivalent to 1.

On the one hand, to reuse a variational expression and configure it differently in different locations seems to suggest a dynamically determined selection operation. For example, with a dynamic interpretation, the following expression would be equivalent to 3+4.

macro v := (**dim** $B\langle b_1, b_2\rangle$ **in** $B\langle 3, 4\rangle$) **in**
(**select** $B.b_1$ **from** v) + (**select** $B.b_2$ **from** v)

Although this expression could be equivalently represented without the need for dynamic scoping, as shown below, this representation is more brittle since the selections are not localized to the places where the variational expression is reused.

select $B.b_1$ **from**
select $B.b_2$ **from**
macro v := (**dim** $B\langle b_1, b_2\rangle$ **in** $B\langle 3, 4\rangle$) **in** v+v

On the other hand, dynamic scoping is risky since dimensions can be unexpectedly captured by selections. This is similar to the issue of macro hygiene in metaprogramming systems [Kohlbecker et al., 1986]. For example, in the following expression where v is defined far away from the selection, the author likely intended the selection to affect only the second declaration of dimension C.

$\textbf{macro}\ v := (\textbf{dim}\ C\langle c_1, c_2\rangle\ \textbf{in}\ C\langle 1, 2\rangle)\ \textbf{in}$
$\dots$
$\textbf{select}\ C.c_1\ \textbf{from}$
$v + (\textbf{dim}\ C\langle c_1, c_3\rangle\ \textbf{in}\ C\langle 3, 4\rangle)$

This expectation is satisfied by a lexically scoped interpretation of selection, which yields the following equivalent expression.

$\textbf{dim}\ C\langle c_1, c_2\rangle\ \textbf{in}\ C\langle 1, 2\rangle + 3$

However, with dynamic scoping, the dimension exposed by the expression bound to v will be captured by the selection, so the expression is equivalent to the following.

$1 + (\textbf{dim}\ C\langle c_1, c_3\rangle\ \textbf{in}\ C\langle 3, 4\rangle)$

The unwanted dimension capture can be avoided by localizing the selection to the intended dimension, such that v is no longer in scope of the selection. Thus, we can rewrite our original example in the following way, which behaves as expected under the dynamic interpretation.

$\textbf{macro}\ v := (\textbf{dim}\ C\langle c_1, c_2\rangle\ \textbf{in}\ C\langle 1, 2\rangle)\ \textbf{in}$
$\dots$
$v + (\textbf{select}\ C.c_1\ \textbf{from}\ \textbf{dim}\ C\langle c_1, c_3\rangle\ \textbf{in}\ C\langle 3, 4\rangle)$

While unexpected dimension captures are bad, they can be avoided by localizing selections as much as possible. In general, dynamic scoping better supports the reuse of variational components since it allows us to locally configure the component at each point where it is reused. Therefore we choose this interpretation as the basis for our semantics.

$$\begin{array}{rcll} e & ::= & \ldots & \\ & | & \textbf{select}\ D.t\ \textbf{from}\ e & \textit{Selection} \end{array}$$

Figure 5.1: Syntax of selection extension.

5.4 Syntax and Denotational Semantics

At this point, the syntax of **select** expressions is familiar, but for completeness a formal definition is given in Figure 5.1. We also extend the definitions of bound and free dimensions from Section 3.3, choice elimination from Section 3.4, and free variables from Section 4.3, all in the obvious ways. We do not extend the well-formedness property, as we have done for previous extensions, because we introduce a more sophisticated type system in the next section that subsumes this property.

Recall that the semantics basis of the choice calculus is a mapping from decisions to plain variants. The impact of **select** expressions is that they will reduce the number of entries in this mapping. For example, assuming that expression e exposes a single dimension named D, then the domain of the mapping $[\![\textbf{select}\ D.t\ \textbf{from}\ e]\!]$ contains no tag in dimension D, while the range is restricted to the variants in $[\![e]\!]$ whose decisions contain the tag $D.t$.

Based on this intuition and taking into account the design decisions in Section 5.3, the denotational semantics of the selection extension is defined in Figure 5.2. As with the core choice calculus and its previous extensions, the semantics is defined compositionally. This makes selection a modular extension to the choice calculus in the sense that it requires no changes to the existing definition and can be arbitrarily included or not in any variant of the choice calculus.

The denotational semantics of a **select** expression is defined by recursively computing the semantics of its target, then filtering the returned mapping. The filtering step is based on the decision in each entry of the target's semantics, with the help of a recursively defined predicate S. Recall that a

$$[\![\textbf{select } D.t \textbf{ from } e]\!] = \{(\delta \backslash D.t, e') \mid (\delta, e') \in [\![e]\!],\ S(\delta)\}$$

$$S(\varepsilon) = \mathit{true}$$

$$S(D'.t'\,\delta') = \begin{cases} t == t' & \text{if } D == D' \\ S(\delta') & \text{otherwise} \end{cases}$$

Figure 5.2: Denotational semantics of selection extension.

decision δ is a list of dimension-qualified tags. On an empty decision ε, the predicate S is true, reflecting the decision in Section 5.3.1 that selection of an undeclared dimension is idempotent. For the recursive case with $D'.t'\,\delta'$, if D' is the selected dimension, then S is true if t' is the selected tag (note that we use == for dimension and tag equality). If t' is not the selected tag, S is false and the entry will be omitted from the semantics. Otherwise, if D' is not the selected dimension, we recursively examine the tail, δ'.

Finally, for entries that pass the filter defined by S, we remove the tag $D.t$ from the resulting decision. The notation $\delta \backslash D.t$ removes the first occurrence of $D.t$ in decision δ. This operation is defined to be idempotent when δ does not contain such a tag since decisions with no tag in dimension D will also be included in the semantics.

Note that this definition implicitly satisfies the design decisions of folding a selection across object structures (until it is consumed) and mapping a selection across choices, motivated in Section 5.3.2 and Section 5.3.4, respectively. Parallel declarations of a dimension D in an object structure show up in the semantics as multiple tags qualified by D in a single decision. This is because each declaration of D defines an independent dimension that must be selected separately. Since the semantics of **select** matches and removes only the first such tag, this corresponds to eliminating only the leftmost parallel dimension declaration in an object structure. Meanwhile, parallel declarations of a dependent dimension D (in some choice) show up in each each decision in the semantics as at most one tag qualified by D. This is because only one alternative of the enclosing choice will ultimately be included. Thus, matching and removing a single tag corresponds to eliminating all parallel dependent dimension declarations in a choice.

5.5 Configuration Type System

This section presents a type system for the fully extended version of the choice calculus with both reuse constructs and internalized selection. The purpose of the type system is twofold. First, it replaces the well-formedness property defined in Section 3.3.3 and extended in Section 4.3. The type system ensures that all choices are properly bound by a corresponding dimension declaration of the appropriate arity, and that all variable references are bound by a **share** or **macro** construct. Second, the type system tracks the *configuration status* of choice calculus expressions. That is, the type of an expression will tell us which decisions must still be made in order to resolve the variational expression into a plain variant in the object language.

Note that this type system is fundamentally different in purpose and structure from other variational type systems that try to ensure the type safety of object languages in the presence of variation, such as the work of Kästner et al. [2012a] on type checking CPP-annotated C programs, or our own work on inferring types for variational lambda calculus [Chen et al., 2012, 2013], which is briefly summarized in Section 9.1.

The configuration status of a choice calculus expression is captured by a judgment of the form $\Gamma \vdash e : \Delta$, which states that expression e has configuration type Δ in the context of the environment Γ. In Section 5.5.1 we describe the structure and meaning of a configuration type. In Section 5.5.2 we describe the structure of the typing environment and the important properties that a configuration judgment can express. Finally, in Section 5.5.3

$$\begin{array}{rcll} \Delta & ::= & \Phi & \textit{Fully Configured} \\ & | & D\langle t \Rightarrow \Delta, \ldots, t \Rightarrow \Delta \rangle ; \Delta & \textit{Required Decision} \end{array}$$

Figure 5.3: Configuration types.

we present the typing rules that associate configuration types with choice calculus expressions.

5.5.1 Configuration Types

A *configuration type* captures the structure of the dimensions that must still be resolved in order to obtain a plain variant. Alternatively, a configuration type can be viewed as a *model* of the decision space of a choice calculus expression (see Section 2.2). The syntax of configuration types is given in Figure 5.3. The constant Φ represents an expression that is *fully configured* in the sense that there are no exposed dimensions of variation. A plain expression would have configuration type Φ, as would an expression in which every declared dimension is resolved internally by **select** expressions. Exposed dimensions are reflected in configuration types by *required decisions*. A required decision consists of two parts, separated by a semicolon.

- The first part $D\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n \rangle$ represents an exposed dimension D with tags $t_1, \ldots, t_n$. The nested configuration types $\Delta_1, \ldots, \Delta_n$ capture dimensions that are dependent on the selection of the corresponding tag(s) in D.

- The second part Δ captures subsequent dimensions that are independent of dimension D.

As an example, the expression **dim** $A\langle a_1, a_2 \rangle$ **in** $A\langle 1, 2 \rangle$ would have the configuration type $A\langle a_1 \Rightarrow \Phi, a_2 \Rightarrow \Phi \rangle ; \Phi$. This reveals that we have a required decision in dimension A, that neither of A's tags have dependent

$$\Phi \oplus \Delta' = \Delta'$$
$$(D\langle\ldots\rangle;\Delta) \oplus \Delta' = D\langle\ldots\rangle;(\Delta \oplus \Delta')$$

Figure 5.4: Configuration type concatenation.

dimensions, and that after making a selection in A, we have no more decisions left to make. For readability, we omit the dependencies of tags that have none (that is, when $t \Rightarrow \Phi$), and omit the terminating Φ symbol of a sequence of required decisions. Therefore we can write the type of the above expression more succinctly as $A\langle a_1, a_2\rangle$.

As a more complex example, consider the following expression in which three dimensions A, B, and C are embedded into two branches of an object structure, and dimension B is dependent on the selection of $A.a_2$.

$$\prec \mathbf{dim}\ A\langle a_1,a_2\rangle\ \mathbf{in}\ A\langle 1, \mathbf{dim}\ B\langle b_1,b_2\rangle\ \mathbf{in}\ B\langle 2,3\rangle\rangle,$$
$$\mathbf{dim}\ C\langle c_1,c_2\rangle\ \mathbf{in}\ C\langle 4,5\rangle \succ$$

The configuration type of this expression is $A\langle a_1, a_2 \Rightarrow B\langle b_1, b_2\rangle\rangle; C\langle c_1, c_2\rangle$, which reveals the relationships between A, B, and C.

Often in the typing rules we need to express a sequence of two or more configuration types. We use the syntax $\Delta \oplus \Delta'$ to express the sequence of type Δ followed by type Δ'. The $\oplus$ operator is associative and is isomorphic to list concatenation. It is defined in Figure 5.4.

5.5.2 Typing Environment and Important Properties

In the configuration judgment $\Gamma \vdash e : \Delta$, the environment Γ contains two kinds of bindings. The first is the standard mapping from variables to configuration types, written $v : \Delta$. The second kind maps dimension names to a pair of integers used to support the typing of choices. In a context that contains the mapping $D : (n, i)$, a choice in dimension D must have exactly n

$$\Gamma \quad ::= \quad \varnothing \;\mid\; \Gamma,\, D : (n, i) \;\mid\; \Gamma,\, v : \Delta$$

Figure 5.5: Configuration typing environment.

alternatives, and the ith alternative is considered to have been selected. The formal definition of Γ is given in Figure 5.5. In the typing rules, we use the syntax $v : \Delta \in \Gamma$ to lookup the type of v in Γ, and likewise use $D : (n, i) \in \Gamma$ to lookup the values of n and i associated with D.

In addition to revealing the decision structure of an expression through its configuration type, there are two significant properties that a typing judgment can express.

1. The judgment $\varnothing \vdash e : \Delta$ expresses the property that e is *well formed*. Restated, a well-formed expression is one that is well typed in the empty environment. Recall from Section 3.3.3 and Section 4.3 that a well-formed expression contains no free variable reference and every choice $D\langle e_1, \ldots, e_n\rangle$ is bound by a corresponding dimension **dim** $D\langle t_1, \ldots, t_n\rangle$.

2. The judgment $\Gamma \vdash e : \Phi$ expresses the property that e is *fully configured* in the context of environment Γ. An expression e is fully configured if it exposes no dimension declarations and every free choice $D\langle e_1, \ldots, e_n\rangle$ is selected by a corresponding entry $D : (n, i)$ in Γ. Note that e may still contain unexpanded **macro** or **share** constructs.

Of course, these two properties can be expressed together by the judgment $\varnothing \vdash e : \Phi$, which means that the expression is both fully configured and well formed. In this case, the semantics of e is a trivial mapping from the empty decision to a single plain variant in the object language.

$$\textsc{C-Leaf}\quad \Gamma \vdash a\prec\succ : \Phi$$

$$\textsc{C-Obj}\quad \frac{\Gamma \vdash e_1 : \Delta_1 \quad \ldots \quad \Gamma \vdash e_n : \Delta_n}{\Gamma \vdash a\prec e_1, \ldots, e_n\succ : \Delta_1 \oplus \ldots \oplus \Delta_n}$$

$$\textsc{C-Dim}\quad \frac{\Gamma,\ D : (n,1) \vdash e : \Delta_1 \quad \ldots \quad \Gamma,\ D : (n,n) \vdash e : \Delta_n}{\Gamma \vdash \textbf{dim}\ D\langle t_1, \ldots, t_n\rangle\ \textbf{in}\ e : D\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n\rangle; \Phi}$$

$$\textsc{C-Chc}\quad \frac{D : (n,i) \in \Gamma \qquad \Gamma \vdash e_i : \Delta_i}{\Gamma \vdash D\langle e_1, \ldots, e_n\rangle : \Delta_i}$$

$$\textsc{C-Sel}\quad \frac{\Gamma \vdash e : \Delta \qquad \Delta \xrightarrow{D.t} \Delta'}{\Gamma \vdash \textbf{select}\ D.t\ \textbf{from}\ e : \Delta'}$$

$$\textsc{C-Shr}\quad \frac{\Gamma \vdash e : \Delta \qquad \Gamma,\ v : \Phi \vdash e' : \Delta'}{\Gamma \vdash \textbf{share}\ v\ := e\ \textbf{in}\ e' : \Delta \oplus \Delta'}$$

$$\textsc{C-Mac}\quad \frac{\Gamma \vdash e : \Delta \qquad \Gamma,\ v : \Delta \vdash e' : \Delta'}{\Gamma \vdash \textbf{macro}\ v := e\ \textbf{in}\ e' : \Delta'}$$

$$\textsc{C-Var}\quad \frac{v : \Delta \in \Gamma}{\Gamma \vdash v : \Delta}$$

Figure 5.6: Configuration typing rules.

5.5.3 Typing Rules

Finally, the typing rules that associate configuration types with choice calculus expressions are given in Figure 5.6.

The typing of object structures is split into two rules, C-Leaf and C-Obj. The C-Leaf rule captures the fact that object structures with no subexpressions are trivially fully configured. Otherwise, in the rule C-Obj, the configuration type is constructed by concatenating the types of each of the subexpressions, reflecting the fact that each subexpression must be configured separately.

Skipping ahead to the rules for the reuse extensions, the rules C-Shr and C-Mac reflect the staging differences between these constructs. For an expression **share** $v\ := e$ **in** e', we configure the bound expression e exactly once, then configure the scope e', which is reflected in the concatenation of their types in the result type. When configuring e', we add v to the typing environment to indicate that it is in scope; however, the type associated with v is Φ since the expression bound to v has already been fully configured. Therefore, each reference to v will not affect the configuration of e'.

For an expression **macro** $v := e$ **in** e', we may have to configure e several times depending on how many times v is referenced in e'. Therefore, the result type is the type of e' extended with the mapping $v : \Delta$. The C-Var rule ensures that each time v is referenced, we will add another instance of Δ to the overall configuration.

The rule C-Dim introduces a new required decision for the declaration of dimension D. The resulting dependent configurations are determined by essentially simulating the selection of each tag in D by typing its scope n times, with the environment extended by $D : (n, i)$ for every i from 1 to n. The C-Chc rule refers to these mappings, using n to ensure that the choice has the correct number of alternatives, and using i to return the type of the corresponding alternative.

For a selection **select** $D.t$ **from** e, the C-Sel rule eliminates required decision(s) in dimension D from the configuration type of its target e. It does this with the help of a type reduction relation $\Delta \overset{D.t}{\rightarrowtail} \Delta'$, defined in Figure 5.7. The reduction relation essentially implements the strategy of mapping over choices and folding over structures, outlined in Section 5.4, but lifted to the type level. In this case, we map over the alternatives of a required decision, and fold over sequences of required decisions until the reduction is consumed. The first rule simply states that fully configured types cannot be further reduced. The second rule implements the fold over sequences by stating that if the reduction of a prefix has no effect, we can attempt to reduce the suffix. The third rule implements the reduction of a matched required decision in D by replacing it with the dependent

$$\Phi \overset{D.t}{\rightharpoondown} \Phi$$

$$\frac{\Delta_1 \overset{D.t}{\rightharpoondown} \Delta_1' \qquad \Delta_2 \overset{D.t}{\rightharpoondown} \Delta_2' \qquad \Delta_1 = \Delta_1'}{\Delta_1 \oplus \Delta_2 \overset{D.t}{\rightharpoondown} \Delta_1 \oplus \Delta_2'}$$

$$\frac{D = D' \qquad t = t_i}{D'\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n \rangle; \Delta \overset{D.t}{\rightharpoondown} \Delta_i \oplus \Delta}$$

$$\frac{D \neq D' \qquad (\Delta_i \overset{D.t}{\rightharpoondown} \Delta_i')^{i:1..n}}{D'\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n \rangle; \Delta \overset{D.t}{\rightharpoondown} D'\langle t_1 \Rightarrow \Delta_1', \ldots, t_n \Rightarrow \Delta_n' \rangle; \Delta}$$

Figure 5.7: Configuration type reduction.

configuration corresponding to t. Finally, the last rule maps a reduction over the alternatives of an unmatched required decision.

5.6 Semantics-Preserving Transformations

In this section we enumerate the semantics-preserving transformation laws involving the extension defined in this chapter. Since selections are resolved at a conceptually later stage than **macro** expressions, and since the target of a **select** is determined dynamically rather than statically, commuting selections is a delicate business. We must be careful not only of respecting which dimension declaration a selection may affect, but also which variable references it may affect. Additionally, while a matching dimension declaration forms a hard lexical boundary, after which we know a selection will be consumed, whether or not a variable reference will consume a selection cannot be determined lexically. For example, consider the following expression.

$$\textsc{Sel}\quad \frac{D \notin BD(e) \qquad FV(e) = \varnothing}{e \equiv \textbf{select}\ D.t\ \textbf{from}\ e}$$

$$\textsc{Sel-Sel}\quad \frac{D \neq D'}{\textbf{select}\ D.t\ \textbf{from}\ (\textbf{select}\ D'.t'\ \textbf{from}\ e) \equiv \textbf{select}\ D'.t'\ \textbf{from}\ (\textbf{select}\ D.t\ \textbf{from}\ e)}$$

Figure 5.8: Selection introduction/elimination and commutation.

$$\begin{array}{l}\textbf{select}\ A.a\ \textbf{from}\\ \prec v, w, \textbf{dim}\ A\langle a, b\rangle\ \textbf{in}\ e_1, e_2 \succ\end{array}$$

If the expression bound to v contains a declaration of dimension A, it will be resolved by the enclosing **select** operation and the selection will be consumed (that is, it will not affect any subsequent subexpressions). However, if the expression bound to v does not contain a declaration of A, then the selection may be consumed by the expression bound to w. If that expression also does not contain a declaration of A, then the selection will resolve the declaration of A in the third subexpression of the structure. Thus without the enclosing context, we cannot determine which subexpression of the structure consumes the selection. However, since we know that A is declared in the third subexpression, we can be sure that the selection does not affect e_2.

Because of this ambiguity, our semantics-preserving transformation laws for the **select** construct must be quite conservative, taking into account all of the subexpressions that could possibly be affected by the selection even though only one of them ultimately will be.

In Figure 5.8 we present a rule for introducing and eliminating selections, along with a rule for commuting two **select** constructs. Applying rule Sel right to left (RL), we can eliminate a selection if we are sure that it cannot

Dim-Sel

$$\frac{D \neq D'}{\textbf{dim } D\langle t^n \rangle \textbf{ in } (\textbf{select } D'.t' \textbf{ from } e) \equiv \textbf{select } D'.t' \textbf{ from } (\textbf{dim } D\langle t^n \rangle \textbf{ in } e)}$$

Chc-Sel

$$D\langle (\textbf{select } D.t \textbf{ from } e_i)^{i:1..n} \rangle \equiv \textbf{select } D.t \textbf{ from } D\langle e^n \rangle$$

Figure 5.9: Choice and dimension commutation rules.

affect the target expression. The rule Sel-Sel reveals that we can arbitrarily commute **select** constructs as long as the two selections are in different dimensions.

In Figure 5.9 we present rules for commuting selections with choices and dimensions. These are the significant rules needed to achieve the normal forms described in Section 3.5.3. Observe in rule Dim-Sel that we can commute a selection and a dimension declaration as long as the two constructs refer to dimensions with a different name, since otherwise the dimension would be captured by (when applied RL) or escape (LR) the **select** operation.

Rule Chc-Sel describes the commutation of selections and choices. Observe that because selections are mapped across the alternatives of a choice, the same selection must appear in every alternative of a choice in order to be factored out. When such a selection does not appear in every alternative, we may be able to setup the transformation by introducing new selections with a LR application of the rule Sel.

Finally, Figure 5.10 describes the commutation of selections and object structures. This transformation is broken into two rules, which implement the constraints described at the beginning of this section. The rules are perhaps best understood when applied LR. The rule Obj-Sel-1 addresses the case where the selection is applied to a subexpression that explicitly declares

Obj-Sel-1

$$\frac{D \in BD(e_i) \qquad (D \notin BD(e_j))^{j:1..i-1} \qquad (FV(e_j) = \varnothing)^{j:1..i-1}}{a\prec e^n[i : \textbf{select } D.t \textbf{ from } e_i]\succ \equiv \textbf{select } D.t \textbf{ from } a\prec e^n\succ}$$

Obj-Sel-2

$$\frac{(D \notin BD(e_j))^{j:1..i-1,i+1..n} \qquad (FV(e_j) = \varnothing)^{j:1..i-1,i+1..n}}{a\prec e^n[i : \textbf{select } D.t \textbf{ from } e_i]\succ \equiv \textbf{select } D.t \textbf{ from } a\prec e^n\succ}$$

Figure 5.10: Commuting selection and object structures.

the dimension D. In this case, e_i will definitely consume the selection, so we can factor the selection out of the object structure as long as every previous subexpression $e_1, \ldots, e_{i-1}$ contains no declarations of D and no free variables (which might later be replaced by an expression that declares D). The rule Obj-Sel-2 addresses the case where it is unclear whether e_i will consume the selection. In this case, the transformation is only semantics preserving if it is impossible for the selection to affect *any* of the other subexpressions.

Chapter 6 – Compositional Choice Calculus

As described in Chapter 2, in general, there are three ways to encode variability in software: the *annotative*, *compositional*, and *metaprogramming* approaches. The choice calculus as presented so far is an example of an annotative approach. However, these approaches excel at capturing different kinds of variation and have complementary strengths and weaknesses. Therefore we would sometimes like to incorporate aspects of each of these approaches into a single variation representation.

This chapter will focus mostly on the advantages of combining the annotative and compositional approaches, though the language we devise will have some metaprogramming capabilities as well. The complementary nature of the annotative and compositional approaches is most evident when considering how to represent overlapping variability in multiple dimensions, called *feature interactions* [Prehofer, 1997]. Compositional approaches excel when interactions are widespread and regular, while annotative representations are best suited for a small number of irregular interactions. The trade-offs involved will be illustrated in Section 6.1. Kästner, Apel, and their collaborators have explored these trade-offs in depth [Kästner and Apel, 2008, 2009, Kästner et al., 2008a, 2009a,c] and identified the need for a way "to combine annotation-based and composition-based approaches in a unified and efficient framework" [Kästner and Apel, 2009].

In this chapter, we will present a variant of the choice calculus, the *compositional choice calculus* (CCC), that generalizes and unifies the compositional and annotative approaches to representing variation. Section 6.2 demonstrates how we can interleave both strategies as needed, reducing feature interactions to their inherent complexity and avoiding complexity introduced by bias in the representation. This version of the choice calculus is more powerful than simply adding annotative variation to compositional components, however. In Section 6.3, we introduce an abstraction construct that extends the calculus into a variation metaprogramming system. The combination of annotative, compositional, and metaprogramming approaches leads to new ways of organizing variation in software and supports the definition of high-level variation abstractions.

The syntax of CCC is defined in Section 6.4, and a denotational semantics for the language is given in Section 6.5. The semantics is interesting because it ensures the hygiene property [Kohlbecker et al., 1986] of variation abstractions through a novel compositional semantics definition, rather than by the traditional renaming strategy.

In Section 6.6 we formally analyze the local expressiveness [Felleisen, 1991] of CCC relative to compositional and annotative representations. This demonstrates that CCC is more locally expressive than either approach alone, and also more expressive than a simple union of the two. Throughout the chapter we provide examples that demonstrate how CCC alleviates the feature interaction problem, can be used to define variation abstractions, and supports the generation and organization of variational structures.

The next section provides the necessary background to motivate the design of the calculus. To make the discussion more concrete, we couch it in terms of *feature-oriented software development* (FOSD), but the representation is not limited to this context. Some of the high-level background material is repeated from Section 2.3, but we provide concrete examples here that will be reused throughout the chapter.

6.1 Motivation

FOSD addresses the classic problems of structured software construction and reuse by decomposing a system into the individual *features* it provides and by making it possible to refer to and manipulate these features directly. This strategy is useful for creating massively variable software. By adding a program generation step in which individual features can be selectively included or excluded, a *software product line* (SPL) of distinct but related programs can be produced [Apel and Kästner, 2009].

Variability is expressed in FOSD at two distinct levels. *Feature modeling* describes the high-level relationships between conceptual features in the problem domain [Kang et al., 1990]. *Feature implementation* associates conceptual features with the code and other artifacts that realize them in the solution domain. The compositional and annotative approaches to representing variability can be viewed as two different approaches to feature implementation.

6.1.1 Compositional Approaches

As described in Section 2.3, compositional approaches are motivated by traditional software engineering pursuits like separation of concerns and stepwise refinement. They attempt to modularize each feature, separating its code and data from other features and from the *base program*, which contains no features (or only essential, non-variational features). To do this, they often rely on a language's native modularization mechanisms, such as classes and subclasses in object-oriented languages, augmented with other abstraction mechanisms like mixins or aspects. More generally, the compositional view considers a feature something that can be *applied to* or *composed with* a program in order to produce a new program incorporating the feature.

Figure 6.1 shows a simple SPL in the compositional style. This running example is based on an example from Liu et al. [2006]. The base program $\underline{b}$ is a simple integer buffer written in Java. Note that we use underlined names to indicate plain (non-variational) expressions in an object language, such as Java. We add to this an optional logging feature $\underline{l}$, implemented as an aspect in the AspectJ language [Kiczales et al., 2001]. The aspect adds a `log` method to the `Buffer` class and inserts a call to this method before the execution of every method in `Buffer`. Thus, our SPL has two products, the basic buffer $\underline{b}$

```
class Buffer {
  int buff = 0;
  int get() {
    return buff;
  }
  void set(int x) {
    buff = x;
  }
}
```

a. Base program, $\underline{b}$.

```
aspect Logging {
  void Buffer.log() {
    print(buff);
  }
  before(Buffer t) :
      target(t) &&
      execution(*) {
    t.log();
  }
}
```

b. Logging feature, $\underline{l}$.

Figure 6.1: A small integer buffer SPL.

and the buffer with the logging feature added, obtained by applying $\underline{l}$ to $\underline{b}$, which we write $\underline{l} \bullet \underline{b}$, following the style of Apel et al. [2008b].

In Figure 6.2, we implement two possible undo features as *class refinements* in the Jak language of the AHEAD Tool Suite [Batory et al., 2004]. When a refinement is applied to a class, new data members and methods in the refinement are added to the class and existing methods are overridden, similar to traditional inheritance. Each of the refinements in Figure 6.2 modifies the `Buffer` class by adding a new data member, adding a new method `undo()`, and overriding the existing `set` method to incorporate some new functionality (the statement beginning with `Super` calls the overridden method). The $\underline{u_o}$ feature adds the ability to undo one previous change to the buffer, while $\underline{u_m}$ adds the ability to undo arbitrarily many changes. Now we can, for example, generate the product $\underline{l} \bullet \underline{u_o} \bullet \underline{b}$, which is an integer buffer with logging and one-step undo.

Note that we have used three different object languages in the creation of this example: $\underline{b}$ is a Java class, $\underline{l}$ is an aspect in AspectJ, and $\underline{u_o}$ and $\underline{u_m}$ are

```
refines class Buffer {
  int back = 0;
  void set(int x) {
    back = buff;
    Super(int).set(x);
  }
  void undo() {
    buff = back;
  }
}
```

a. Undo-one feature, $\underline{u_o}$.

```
refines class Buffer {
  Stack stack = new Stack();
  void set(int x) {
    stack.push(buff);
    Super(int).set(x);
  }
  void undo() {
    buff = stack.pop();
  }
}
```

b. Undo-many feature, $\underline{u_m}$.

Figure 6.2: Class refinements implementing undo features.

Jak class refinements. This reveals that the feature composition operator $\bullet$ is overloaded—the operation it performs depends on the types of its arguments. When we write $\underline{l} \bullet \underline{b}$, the operator represents aspect weaving [Elrad et al., 2001], while in $\underline{u_o} \bullet \underline{b}$ it represents class refinement. This makes it possible to extend the compositional approach to new object languages and artifact types by simply adding new instances of the $\bullet$ operator [Batory et al., 2004, Apel et al., 2008b].

6.1.2 Annotative Approaches and the Choice Calculus

Annotative approaches represent variation in-place, by directly marking the code to be conditionally included if the corresponding features are selected. The choice calculus as presented in Chapter 3 is a formal language for representing annotative variation. We call the variant introduced in this chapter the *compositional* choice calculus because it adds compositional functionality

```
dim Log⟨yes,no⟩ in
dim Undo⟨one,many,none⟩ in
class Buffer {
  int buff = 0;
  Undo⟨int back = 0,Stack stack = new Stack(),○⟩;
  int get() { return buff; }
  int set(int x) {
    Log⟨print(buff+"->"+x),○⟩;
    Undo⟨back = buff,stack.push(buff),○⟩;
    buff = x;
  }
  Undo⟨void undo() {
          Log⟨print(back+"<-"+buff),○⟩;
          buff = back;
        },
        void undo() {
          Log⟨print(stack),○⟩;
          buff = stack.pop();
        },○⟩
}
```

Figure 6.3: Buffer with annotated logging and undo features.

to this annotative core. Likewise, when the distinction is significant, we refer to the core choice calculus as the *annotative* choice calculus.

Figure 6.3 shows a version of our integer buffer SPL implemented in the annotative style of the choice calculus. For illustrative purposes, the logging feature differs from the aspect-based implementation in the previous subsection. While the compositional logging feature $\underline{l}$ simply prints the value of the buffer after each method call, the annotative logging feature implemented in Figure 6.3 prints a unique message whenever the value of the buffer changes.

Our example contains two dimensions, *Log* and *Undo*. Choices in the *Log* dimension have two alternatives, corresponding to whether change-logging is included or not. Choices in the *Undo* dimension have three alternatives, corresponding to two possible implementations of the undo feature (undo-one or undo-many) and the case where no undo feature is included. For example, to get an integer buffer with no logging and one-step undo, we can select *Log.no* and *Undo.one*.

Recall from Section 3.3.1 that the choice calculus respects the tree structure of the underlying artifact it varies, ensuring the syntactic correctness of all variants. Formally, each node in the tree is encoded by a constant value a from the object language and a possibly empty list of subexpressions, written $a\prec e_1, \ldots, e_n\succ$. For example, we can represent the buffer's `get` method as $\texttt{get}\prec\texttt{()},\texttt{return}\prec\texttt{buff}\succ\succ$. We rarely show this tree structure explicitly, preferring to embed the notation directly in the concrete syntax of the object language, for readability. However, some of the later formalism will make use of the formal representation of structure nodes.

All of the choices in our example contain the empty expression $\circ$ as their last alternative. This is an element of the object language, not of the choice calculus itself. This is important because it preserves the guarantee of syntactic correctness—such a value can only be included as an alternative where it is syntactically valid in the object language. For example, the choice calculus expression $\texttt{1+}D\langle\texttt{2},\circ\rangle$ is invalid since $\circ$ is not syntactically correct at the position of the choice in the surrounding Java expression. However, the choice calculus expression $\texttt{1+}D\langle\texttt{2},\texttt{x}\rangle$ *is* valid. Even though the literal `2` and variable `x` are different syntactic categories, both alternatives are syntactically correct at the position of the choice.

6.1.3 Representing Feature Interactions

The salient problem in FOSD is detecting, resolving, and managing the interactions of a huge number of conditionally included features [Kästner et al., 2009c]. This is a large problem that spans all stages of the software life cycle. Here we consider only the much smaller subproblem of *representing* intended feature interactions in a way that is structured and manageable.

Interactions are represented quite differently in the annotative and compositional approaches to feature implementation. In the annotative approach exemplified by the choice calculus, interactions appear as nested choices. For example, the *Log* choices inside of the *Undo* choice in Figure 6.3 capture the interaction between the undo and logging features. This way of representing feature interactions is simple and explicit. It is best suited for interactions between a small number of features, where each interaction must be handled uniquely.

However, many interactions are regular and cut across many features. Generic logging is a classic example. For every feature that adds a new method, we must also define its interaction with the logging feature. This quickly leads to maintenance issues with even a small number of features. The compositional approach addresses this problem through the creation of new abstraction mechanisms. For example, the representation of the logging feature $\underline{l}$ as an aspect in Figure 6.1.b demonstrates how cross-cutting, regular interactions can be modularized by introducing a new kind of artifact, in this case, aspects. As long as we apply $\underline{l}$ after including the undo feature, both the base program and undo feature's methods will be extended accordingly.

The logging feature as implemented in the annotative example in Figure 6.3 is less regular, however. Its interaction with the undo feature is messy since it prints a different message depending on which variant of the undo feature we select. Such irregular interactions are trivial in the annotative approach but require special consideration in the compositional approach.

```
refines class Buffer {
  void set(int x) {
    print(buff+"->"+x);
    Super(int).set(x);
  }
}
```

a. Add logging to $\underline{b}$, $\underline{l_b}$.

```
refines class Buffer {
  void undo() {
    print(back+"<-"+buff);
    Super().undo();
  }
}
```

b. Add logging to $\underline{u_o}$, $\underline{l_{u_o}}$.

```
refines class Buffer {
  void undo() {
    print(stack);
    Super().undo();
  }
}
```

c. Add logging to $\underline{u_m}$, $\underline{l_{u_m}}$.

Figure 6.4: Modularized feature interactions.

A solution to the problem of representing irregular interactions in the compositional approach is described by Liu et al. [2006] and demonstrated in Figure 6.4. Essentially, we split the representation of the logging feature into several smaller refinements. The refinement $\underline{l_b}$ adds logging to the base program, while refinements $\underline{l_{u_o}}$ and $\underline{l_{u_m}}$ add logging to the undo-one and undo-many features, respectively. The $\underline{l_{u_o}}$ and $\underline{l_{u_m}}$ refinements directly capture the interaction of the logging and undo features. Now we can generate a program with only the logging feature by applying $\underline{l_b} \bullet \underline{b}$, and add to this the undo-one feature by applying $\underline{l_{u_o}} \bullet \underline{u_o} \bullet \underline{l_b} \bullet \underline{b}$.

By spreading a feature's implementation across several modules, this solution mortgages some of the benefits of feature modularity promised by the compositional approach. In the worst-case, there can be an exponential explosion of such feature-interaction modules [Liu et al., 2006]

6.2 Integrating the Two Approaches

One of the goals of CCC is to unify the annotative and compositional approaches to feature implementation. Although the complete syntax of the calculus is not presented until Section 6.4, the examples in this section and the next are given in CCC and illustrate how it addresses several challenging variation representation scenarios.

The compositional and annotative approaches to feature implementation are highly complementary. Compositional approaches separate features at the expense of variation granularity and flexibility. Annotative approaches are highly flexible and granular, but do not separate features.

These trade-offs are evident even in our very simple integer buffer SPL. The separated undo features $\underline{u_o}$ and $\underline{u_m}$ in Figure 6.2 can be implemented without changing the base program $\underline{b}$, and $\underline{b}$ can be understood without knowledge of the undo features. These qualities reflect the tenets of step-wise refinement and separation of concerns, respectively, that the compositional approach is founded on. In contrast, the annotative implementation of undo in Figure 6.3 requires direct modification of the base program and clutters its definition with code that is only sometimes relevant.[1] However, the two compositional undo features contain quite a lot of redundant boilerplate code that is not needed in the annotative representation. This complicates the maintenance of the compositional representation; for example, if we change the name of the `set` method in the base program, we must also change its name in both $\underline{u_o}$ and $\underline{u_m}$.

Figure 6.5 presents an obvious compromise, where we annotate a compositional feature implementation. This allows the two variants of the undo

[1] Better user interfaces can alleviate some of the readability concerns through a "virtual separation of concerns" [Kästner and Apel, 2009, Le et al., 2011].

```
dim Undo⟨one, many⟩ in
refines class Buffer {
  Undo⟨int back = 0,Stack stack = new Stack()⟩;
  void set(int x) {
    Undo⟨back = buff,stack.push(buff)⟩;
    Super(int).set(x);
  }
  void undo() {
    buff = Undo⟨back,stack.pop()⟩;
  }
}
```

Figure 6.5: Undo refinement u with annotative variation.

feature to share their common code while retaining separability with respect to the base program. This new annotated refinement u was created by simply merging $\underline{u_o}$ and $\underline{u_m}$, introducing a new dimension *Undo* to capture their differences in synchronized choices.

In fact, it is possible to mechanically derive u from $\underline{u_o}$ and $\underline{u_m}$ using the semantics-preserving transformation laws presented in Section 3.5. We begin with the following choice calculus expression.

$$\textbf{dim}\ \mathit{Undo}\langle \mathit{one}, \mathit{many}\rangle\ \textbf{in}\ \mathit{Undo}\langle \underline{u_o}, \underline{u_m}\rangle$$

Then we maximally factor the commonalities out of $\underline{u_o}$ and $\underline{u_m}$, for example, using the Chc-Obj rule, effectively localizing the differences between the two undo features. Since there is a semantics-preserving transformation from the above expression to u, we can deduce that they are equivalent.

Introducing a top-level dimension to choose between $\underline{u_o}$ and $\underline{u_m}$ suggests the idea of using annotations not only in the code that implements features, but also in the algebra that describes the composition of features into products. The non-commutativity of feature composition often leads to ordering constraints between features at the implementation level that do not exist at the conceptual feature modeling level. For example, the decision of whether to include the logging feature $\underline{l}$ and the undo-many feature $\underline{u_m}$ are conceptually independent, but $\underline{l} \bullet \underline{u_m} \bullet \underline{b}$ and $\underline{u_m} \bullet \underline{l} \bullet \underline{b}$ produce different programs. In the second product, calls to methods in the undo feature will not be logged since these are added only after the logging aspect is woven in. These kinds of implementation-specific ordering constraints make assembling components into products potentially error-prone.

To solve this problem, we can write a choice calculus expression that encodes the ordering constraints and describes all of the products that can be generated. This way, we just select a variant and the product will be composed in the correct order. For example, if we introduce a "dummy" feature *id* such that $id \bullet p \equiv p$ for any p, then we can describe all of the products in our integer buffer SPL with the following expression.

$$(\textbf{dim}\ Log\langle yes, no\rangle\ \textbf{in}\ Log\langle \underline{l}, id\rangle) \bullet$$
$$(\textbf{dim}\ Undo\langle yes, no\rangle\ \textbf{in}\ Undo\langle u, id\rangle) \bullet \underline{b}$$

Note that the u component itself contains variation in a different dimension named *Undo* (with tags *one* and *many*). So if we select *Undo.yes* we must also make a selection in the nested *Undo* dimension to determine which variant of the undo feature we want to include.

We can now obtain all of the products in our SPL by making selections on the above expression. For example, selecting $[Log.no, Undo.yes, Undo.one]$ produces the integer buffer with one-step undo and no logging. If we select *no* in the outer *Undo* dimension, then we do not make a selection in the inner *Undo* dimension since the inner *Undo* dimension in u will not be included.

This solution does not eliminate the ordering constraints between features but rather captures them once-and-for-all alongside constraints imposed by the feature model (for example, that logging and undo are optional). This enables a concise definition of all generable variants, properly composed.

The primary motivation for integrating the annotative and compositional approaches into a single representation is to provide maximal flexibility in representing interactions between features. For example, the interaction of the irregular logging feature and the two alternate undo features requires the two refinements $\underline{l_{u_o}}$ and $\underline{l_{u_m}}$ in Figure 6.4. With an integrated representation, we can combine these refinements in the same way we produced u from $\underline{u_o}$ and $\underline{u_m}$. We could alternatively include the interactions directly in the implementations of $\underline{u_o}$ and $\underline{u_m}$, using annotations. Either option would reduce redundancy in the implementation and the specific choice of which representation to use can be left to the features' implementors.

6.3 Adding Variation Abstractions

The previous section described a straightforward mixture of annotative and compositional approaches to feature implementation. This was enabled by just applying the annotative choice calculus to compositional components and to the feature algebras used to assemble these components.

In this section we further integrate the two approaches by introducing an abstraction construct and generalizing feature composition to function application. The result is the compositional choice calculus. These changes support reuse, the minimization of redundancy, and extend the choice calculus with some basic metaprogramming capabilities. This section motivates these extensions through several examples.

6.3.1 Reusable Optional Wrappers

A bit of unaddressed redundancy in our integer buffer example is the repetition of the `undo` method declaration in the first two alternatives of the *Undo* choice in Figure 6.3. Although the body of the method differs, the declaration is the same, so we would like to abstract this commonality out. We cannot just push the choice into the body of the method, however, because the third alternative (corresponding to *Undo.none*) does not declare the method.

In Chapter 4 we introduced some extensions to the choice calculus that can be used to solve this problem. Using the **share** construct, we can factor the redundancy out as follows. For conciseness, we introduce the variables b_o and b_m to refer to the body of the `undo` method corresponding to the undo-one and undo-many features, respectively.

$$\begin{array}{l} \textbf{share}\ u_{decl} := (\ \textbf{share}\ u_{body} := \mathit{Undo}\langle b_o, b_m, \circ\rangle \\ \qquad\qquad\qquad\ \ \textbf{in}\ \texttt{void undo() \{ } u_{body}\ \texttt{\}}\ ) \\ \textbf{in}\ \mathit{Undo}\langle u_{decl}, u_{decl}, \circ\rangle \end{array}$$

This solution works but is troublingly inelegant. The problem is related to the *optional wrapper* problem described by [Kästner et al., 2008b], and discussed in Section 4.1.3. It describes a variation pattern where an expression is conditionally wrapped in another construct, such as a conditional statement or try-catch block. Since the code shared between variants is a subexpression of the optional wrapper, it is difficult to mark only the wrapper as optional. CIDE handles this pattern by designating certain constructs in the object language as wrappers and treating them specially. The choice calculus's **share** construct is a more general solution that works well for single optional wrappers—for example, we can optionally wrap the expression e in a try-catch block as follows.

$$\textbf{share}\ v := e\ \textbf{in}\ D\langle \texttt{try \{ } v \texttt{ \} catch \{ ... \}}, v\rangle$$

But as our `undo` example demonstrates, it breaks down when we want to reuse the wrapper in multiple alternatives.

In the compositional choice calculus, we split the **share** construct into separate abstraction and application constructs, which we write in the style of lambda calculus as $\lambda v.e$ and $e\ e'$, respectively. This allows us to capture the `undo` method declaration wrapper u_w as follows.

$$\lambda b.\,\texttt{void undo() \{ } b \texttt{ \}} \qquad (u_w)$$

We can then rewrite the optional `undo` method from Figure 6.3 more simply by applying u_w to the two different method bodies within the *Undo* choice, as $Undo\langle u_w\ b_o, u_w\ b_m, \circ\rangle$.

Abstractions are useful for representing all sorts of variation patterns, not just optional wrappers. Unlike the annotative choice calculus's **share** construct, which is expanded only after dimensions and choices are resolved, CCC expressions are evaluated top-down, interleaving β-reduction and dimension elimination as needed (see the semantics definition in Section 6.5). In this way, it is more similar to the **macro** construct, also introduced in Chapter 4, but even more powerful since functions can be passed as arguments to other functions. Therefore, rather than just factoring redundancy, it is possible to programmatically create and manipulate the variation structure (dimensions and choices) of CCC expressions in the language itself. The next subsection gives several examples of variation abstractions that do this.

6.3.2 Variation Metaprogramming

In addition to feature implementation, CCC can also abstract and modularize high-level relationships *between* features. Consider the following higher-order function *opt* that accepts two arguments: f is a function that implements a feature, and b is a base program that f can be applied to.

$$\lambda f.\lambda b.\ \textbf{dim}\ Opt\langle yes, no\rangle\ \textbf{in}\ (Opt\langle f, \lambda x.x\rangle\ b) \qquad (opt)$$

If we select *yes* in the enclosed *Opt* dimension, f will be applied to b, if we select *no*, the identity function will be applied. In other words, this function modularizes the notion of feature optionality. We can take any feature f' and make it optional by applying *opt* f'.

Similarly, the following function modularizes the alternative relationship between two features f_1 and f_2.

$$\lambda f_1.\lambda f_2.\lambda b.\ \mathbf{dim}\ \mathit{Alt}\langle \mathit{fst}, \mathit{snd}\rangle\ \mathbf{in}\ (\mathit{Alt}\langle f_1, f_2\rangle\ b) \qquad (\mathit{alt})$$

Exactly one of the two features will be applied to b, depending on our selection in the dimension *Alt*.

These examples illustrate how CCC can be used to directly relate the implementations of features with their high-level organization in feature models, providing a link between the problem and solution domains. As a final demonstration of the potential of this approach, consider the following expression *arb*.

$$\lambda f.\lambda b.(\lambda y.y\ y)\ (\lambda r.\mathbf{dim}\ \mathit{Arb}\langle \mathit{yes}, \mathit{no}\rangle\ \mathbf{in}\ \mathit{Arb}\langle f\ (r\ r), b\rangle) \qquad (\mathit{arb})$$

This function accepts a feature f and a program b, then recursively applies f to b an arbitrary number of times. Each time the *yes* tag is selected from *Arb*, a new copy of the *Arb* dimension is generated and another decision must be made. The recursion will terminate only when *no* is finally selected. Thus, *arb* represents a variational fixed point combinator with an interactive terminating condition. This variational model of computation as an interaction between functions and decisions could have applications far beyond FOSD.

6.4 The Compositional Choice Calculus

The syntax of the compositional choice calculus is given in Figure 6.6. The first three constructs are from the annotative choice calculus, as presented in Chapter 3. The first construct encodes the tree-structure of the object language, choices introduce variation points within that structure, and di-

$$
\begin{array}{rcll}
e & ::= & a\prec e,\ldots,e\succ & \textit{Structure} \\
& | & \textbf{dim}\ D\langle t,\ldots,t\rangle\ \textbf{in}\ e & \textit{Dimension} \\
& | & D\langle e,\ldots,e\rangle & \textit{Choice} \\
& | & \lambda v.e & \textit{Abstraction} \\
& | & e\ e & \textit{Application} \\
& | & v & \textit{Reference}
\end{array}
$$

Figure 6.6: Syntax of the compositional choice calculus.

mensions scope and synchronize choices and organize the variation space. The next three constructs, borrowed from the lambda calculus, extend the choice calculus with the separable, dynamic metaprogramming constructs introduced in the previous section.

Although the concrete syntax is the same, the interpretation of the lambda calculus constructs in CCC is fundamentally different than in the variational lambda calculus (VLC), as introduced in Section 3.3.1 and used in our work on variation typing [Chen et al., 2012, 2013]. In VLC, the lambda calculus is the object language, whereas in CCC abstractions, applications, and variable references are part of the metalanguage (see Section 2.1). The semantics of VLC involves no computation—dimensions and choices describe static variability in the object language of lambda calculus expressions. In contrast, the functional constructs in CCC describe computations involving dimensions and choices that, when evaluated, yield variants in some other object language.

Note that we do not syntactically restrict the LHS of applications to abstractions. Obviously, we want to allow variable references here since variables can be bound to functions, but in fact we can extend application to all other syntactic categories as well—this is the key to unifying the annotative, compositional, and metaprogramming approaches.

Application can be viewed as a generalization of the overloaded feature composition operator • from Section 6.1.1. As a special case, when we apply

App-Dim-l

$$\frac{D \notin FD(e')}{(\textbf{dim } D\langle t_1, \ldots, t_n\rangle \textbf{ in } e)\ e' \equiv \textbf{dim } D\langle t_1, \ldots, t_n\rangle \textbf{ in } e\ e'}$$

App-Chc-l

$$D\langle e_1, \ldots, e_n\rangle\ e' \equiv D\langle e_1\ e', \ldots, e_n\ e'\rangle$$

App-Chc-r

$$e\ D\langle e'_1, \ldots, e'_n\rangle \equiv D\langle e\ e'_1, \ldots, e\ e'_n\rangle$$

Abs-Chc

$$\lambda v.D\langle e_1, \ldots, e_n\rangle \equiv D\langle \lambda v.e_1, \ldots, \lambda v.e_n\rangle$$

Figure 6.7: New equivalence laws for ccc.

two plain expressions $\underline{e}\ \underline{e}'$, we defer to the instance of $\bullet$ determined by the types of $\underline{e}$ and $\underline{e}'$. This is the critical link between the compositional choice calculus and the compositional approach to feature implementation.

The other cases are enumerated and defined formally in the semantics in Section 6.5, but the idea is simple. When an application contains a dimension or choice on the LHS, the result can be obtained by first distributing the application across the dimension or choice, then recursively considering the subexpressions. This suggests the new semantics-preserving transformations laws shown in Figure 6.7, which extend the equivalence relation we have developed throughout this thesis.

The law App-Dim-l distributes across dimension declarations in the LHS of an application. Recall that the function $FD(e)$ returns the set of free dimensions in e. Since we change the scope of the dimension D, the premise of App-Dim-l prevents the capture of choices in e'. The laws App-Chc-l and

App-Chc-r distribute across choices in the LHS and RHS, respectively, of an application. Note that although the commutation of choices and applications is symmetric, the commutation of dimensions and applications is not. That is, there no law App-Dim-r. This is because dimension declarations in the RHS of an application can be duplicated during β-reduction, producing conceptually distinct dimensions.

Finally, the law Abs-Chc straightforwardly commutes abstractions and choices. There is no law for commuting abstractions and dimensions since dimension declarations inside of abstractions can be duplicated. The *arb* example in Section 6.3.2 is one such example.

To demonstrate the evaluation of an expression, consider a variational program that optionally applies the undo-one feature $\underline{u_o}$ to the basic integer buffer $\underline{b}$, then applies the logging feature $\underline{l}$. This variational program can be represented by the following ccc term.

$$\underline{l}\ (opt\ \underline{u_o}\ \underline{b})$$

If we expand *opt* and perform β-reduction twice to consume its arguments, we get the following expression in which the variability is more obvious.

$$\underline{l}\ (\textbf{dim}\ Opt\langle yes, no\rangle\ \textbf{in}\ Opt\langle \underline{u_o}, \lambda x.x\rangle\ \underline{b})$$

Selecting *Opt.yes* yields an integer buffer with both the undo-one and logging features included:

$$\underline{l}\ (\underline{u_o}\ \underline{b}) \equiv \underline{l} \bullet \underline{u_o} \bullet \underline{b}$$

While selecting *Opt.no* yields an integer buffer with only logging:

$$\underline{l}\ ((\lambda x.x)\ \underline{b}) \equiv \underline{l} \bullet \underline{b}$$

While the reduction process described above is rather ad hoc, it captures the essence of the semantics of ccc. Intuitively, the meaning of a ccc expression is the total set of plain variants it represents and the decisions that lead to those variants, just as in the annotative choice calculus. We formalize the relationship between decisions and variants in the next section.

6.5 Denotational Semantics

A CCC expression encodes a decision space, where dimensions describe the decisions that must be made, and choices and computations determine the results of those decisions. Just as we do for the annotative choice calculus, we define the denotation of a CCC expression to be a mapping from decisions to the plain artifacts those decisions produce. Determining this mapping is complicated by the fact that function applications can duplicate and remove dimension declarations, so we cannot statically determine the decisions that must be made for a given CCC expression.

Conceptually, evaluating a CCC expression proceeds in normal order (outermost, leftmost first) and consists of alternating between (1) reducing application nodes and (2) eliminating dimension nodes. This leads to an interactive view of evaluation where we reduce as far as we can, present a decision point to some client, then proceed reducing based on its response. For the purpose of defining a denotational semantics, we simulate this by building a (potentially infinite) mapping that represents all possible decision sequences and the plain expressions they ultimately produce.

In the next subsection we describe two challenges for the definition of a denotational semantics for CCC, then sketch the solution at a high-level. The rest of this section will develop the formal definition of the semantics in several steps.

6.5.1 Challenges for a Formal Semantics

In Section 6.4 we resolved function application with standard lambda calculus β-reduction. Because β-reduction relies on variable substitution, however, we can run into problems when choices are substituted across dimension scopes. Consider the following expression, which contains declarations of two different dimensions A.

$$(\lambda f.\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ f\ A\langle 1,2\rangle)\ (\lambda x.\textbf{dim}\ A\langle c,d\rangle\ \textbf{in}\ x)$$

By applying β-reduction twice—first to the redex at the top of the expression, then to the new redex created by the first reduction—we get the following expression in which the choice in A is bound to the declaration of A with tags c and d, rather than its original dimension with tags a and b.

$$\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ \textbf{dim}\ A\langle c,d\rangle\ \textbf{in}\ A\langle 1,2\rangle$$

Recall from Section 4.4 that we call this phenomenon *choice capture*, and it is highly undesirable since it breaks the static lexical scoping of dimension names. The situation is analogous to the hygiene issue in metaprogramming systems [Kohlbecker et al., 1986].

Just as we cannot arbitrarily expand **macro**-expressions, we cannot arbitrary β-reduce redexes. By the same argument as in Section 4.4, however, we need not worry about choice capture if we (1) assume that expressions are well-formed, and (2) reduce expressions in normal order, interleaving dimension elimination and β-reduction. This is because choice capture can only occur when the RHS of a redex contains a locally free choice. Consider our example after the first reduction.

$$\textbf{dim}\ A\langle a,b\rangle\ \textbf{in}\ (\lambda x.\textbf{dim}\ A\langle c,d\rangle\ \textbf{in}\ x)\ A\langle 1,2\rangle$$

The RHS of the new redex is a free choice from the local perspective of the redex. However, since the expression is well-formed, it is not free in the larger expression. Furthermore, since it is not free in the larger expression, the evaluation of its corresponding dimension declaration will occur *before* the redex is encountered in a normal order evaluation. Therefore, the locally free choice will be eliminated before the redex is resolved.

The other major challenge of a formal semantics for CCC is that we want to directly reuse existing compositional feature implementation tools (such as AHEAD) in a mixed annotative/compositional setting. Therefore, our semantics should make use of the overloaded $\bullet$ operator to implement feature composition, but this operator must *only* be applied to plain expressions. If we satisfy this constraint in the semantics, we can substitute in any off-the-shelf tool(s) to implement $\bullet$, depending on the abstractions and artifact types we care about. However, this constraint should not be reflected at the syntactic level. That is, given a feature f and a base program b, we must be able to evaluate $f\ b$ even if f and b both contain annotations.

With these constraints in mind, the semantics of application is defined by first computing the *partial semantics* of each subexpression, which resolves each subexpression into a mapping from decisions to values, where a value is either a function or a plain expression. These mappings are then combined with the help of a partial semantics composition function $\bowtie$, which performs a pairwise combination of values, invoking either β-reduction or the $\bullet$ operator depending on the type of the values. By computing the partial semantics of the expressions separately, then combining the results, we can ensure that we both preserve the lexical scoping of dimension names and invoke the $\bullet$ operator only on plain expressions.

In the rest of this section we will build up the formal semantics definition in three steps. Section 6.5.2 describes the process of dimension and choice elimination, which is based on the semantics definition of the annotative choice calculus, presented in Section 3.4. Section 6.5.3 defines the structure of a partial semantics mapping and defines the $\bowtie$ operator for composing them. Finally, Section 6.5.4 defines how to compute the partial semantics of a CCC expression.

$$
\begin{aligned}
\lfloor a{\prec}e_1,\ldots,e_n{\succ}\rfloor_{D.i} &= a{\prec}\lfloor e_1\rfloor_{D.i},\ldots,\lfloor e_n\rfloor_{D.i}{\succ} \\
\lfloor \mathbf{dim}\ D'\langle t^n\rangle\ \mathbf{in}\ e\rfloor_{D.i} &= \begin{cases} \mathbf{dim}\ D'\langle t^n\rangle\ \mathbf{in}\ e & \text{if } D = D' \\ \mathbf{dim}\ D'\langle t^n\rangle\ \mathbf{in}\ \lfloor e\rfloor_{D.i} & \text{otherwise} \end{cases} \\
\lfloor D'\langle e_1,\ldots,e_n\rangle\rfloor_{D.i} &= \begin{cases} \lfloor e_i\rfloor_{D.i} & \text{if } D = D' \\ D'\langle\lfloor e_1\rfloor_{D.i},\ldots,\lfloor e_n\rfloor_{D.i}\rangle & \text{otherwise} \end{cases} \\
\lfloor \lambda v.e\rfloor_{D.i} &= \lambda v.\lfloor e\rfloor_{D.i} \\
\lfloor e\ e'\rfloor_{D.i} &= \lfloor e\rfloor_{D.i}\ \lfloor e'\rfloor_{D.i} \\
\lfloor v\rfloor_{D.i} &= v
\end{aligned}
$$

Figure 6.8: Extended choice elimination.

6.5.2 Dimension and Choice Elimination

Recall from Section 3.4 that a decision δ is represented by a sequence of qualified tags, where a qualified tag $q = D.t$ is a tag t prefixed by its dimension D. We use ε to represent the empty decision, and use adjacency to prepend a tag q to an existing decision δ, as in $q\delta$, and to concatenate two decisions δ and δ', as in $\delta\delta'$.

The order that tags are selected from an expression is determined by the order that dimension declarations are encountered during a *normal-order* evaluation strategy. This ordering constraint is needed since function applications can eliminate dimension declarations, or multiply a single declaration into many independent dimensions. An example of this phenomenon is demonstrated by the function *arb* in Section 6.3.2. Therefore, we must not eliminate dimensions too early, or these effects will be lost.

Tag selection thus consists of (1) identifying the next dimension to be selected from, if any, (2) selecting a tag, (3) eliminating the choices bound by that dimension, and then (4) eliminating the dimension declaration itself.

$$\begin{array}{rcll} \varphi & ::= & \underline{e} & \textit{Plain Expression} \\ & | & (\lambda v.e, \rho) & \textit{Closure} \end{array}$$

Figure 6.9: Partial semantics values.

When computing the semantics, each of these steps but the third is handled by the partial semantics function defined in Section 6.5.4. Recall that step (3) is called choice elimination, and is defined as follows. Given a dimension declaration **dim** $D\langle t_1, \ldots, t_n \rangle$ and a selected tag t_i, we write $\lfloor e \rfloor_{D.i}$ to replace every free choice $D\langle e_1, \ldots, e_n \rangle$ in e with its ith alternative, e_i. Figure 6.8 extends the definition of choice elimination from Section 3.4 to CCC. The first three cases are as before, while the new cases simply propagate the selection to their subexpressions, if applicable.

6.5.3 Composing Partial Semantics

The partial semantics, S, of an expression is a mapping from decisions to *values*, where a value φ is either a plain expression, or a *closure*. The representation of values is defined in Figure 6.9. A closure is a CCC abstraction, $\lambda v.e$, paired with its static environment, ρ. Somewhat unusually, the environment stored in a closure does not map variables to plain values, but rather variables to partial semantics mappings. That is, ρ has type $v \to S$. We use the notation $(v, S) \oplus \rho$ to map variable v to partial semantics S in environment ρ.

To compute the partial semantics of an expression e within the environment ρ, we write $V_\rho(e)$. Thus, V has type $(\rho, e) \to S$. The implementation of this function will be given in the next subsection.

Given expressions e_l and e_r in environment ρ, we can compute the semantics of the application $e_l\ e_r$ by computing the partial semantics of e_l and e_r individually, then composing the results. We use the operator $\bowtie$ to represent the composition of two partial semantics mappings. That is, assuming $V_\rho(e_l) = S_l$ and $V_\rho(e_r) = S_r$, then the semantics of $e_l\ e_r$ is $S_l \bowtie S_r$. Since e_l

$$
\begin{aligned}
&S_l \bowtie S_r = \bigcup \{(\delta_l, \varphi_l) \triangleleft S_r \mid (\delta_l, \varphi_l) \in S_l\} \\
&\quad \textbf{where} \\
&\qquad\qquad (\delta_l, \underline{e}'_l) \triangleleft S_r = \{(\delta_l \delta_r, \underline{e}'_l \bullet \underline{e}'_r) \mid (\delta_r, \underline{e}'_r) \in S_r\} \\
&\qquad (\delta_l, (\lambda v.e, \rho)) \triangleleft S_r = \{(\delta_l \delta', e') \mid (\delta', e') \in V_{(v,S_r) \oplus \rho}(e)\}
\end{aligned}
$$

Figure 6.10: Partial semantics composition.

and e_r can be arbitrarily large CCC expressions representing potentially many variants each, partial semantics composition can be viewed as the pairwise application of every variant of e_l to every variant of e_r.

The composition operation is formally defined in Figure 6.10. The operation is defined in terms of a helper function $\triangleleft$ that applies each entry in S_l to the partial semantics S_r. That is, we map the partially applied function $(\cdot \triangleleft S_r)$ across the entries in S_l, then take the union of the results.

Each entry in S_l consists of a decision δ_l and a value φ_l. There are two cases to consider based on the structure of φ_l. These are reflected by the two cases of the $\triangleleft$ operation.

1. If the value φ_l is a plain expression $\underline{e}'_l$, then we require that every value $\varphi_r \in rng(S_r)$ must also be a plain expression $\underline{e}'_r$. Now we can compose $\underline{e}'_l$ with each $\underline{e}'_r$ using the object language composition operator $\bullet$. We must also concatenate the decision that produced $\underline{e}'_l$ with the decision that produced $\underline{e}'_r$ to create the decision that produces the combined expression. Note that if φ_l is plain but there is a value $\varphi_r \in rng(S_r)$ that is *not* plain, then the semantics is undefined.

2. Otherwise, φ_l is a closure $(\lambda v.e, \rho)$. In this case, we simulate β-reduction by adding the mapping (v, S_r) to the environment and computing the partial semantics of the body of the abstraction, e. We then iterate over the results, adding each to our resulting partial semantics.

$$\begin{aligned}
V_\rho(v) &= \rho(v) \\
V_\rho(e\ e') &= V_\rho(e) \bowtie V_\rho(e') \\
V_\rho(\lambda v.e) &= \{(\varepsilon, (\lambda v.e, \rho))\} \\
V_\rho(a\prec\succ) &= \{(\varepsilon, a\prec\succ)\} \\
V_\rho(a\prec e_1, \ldots, e_n\succ) &= \{(\delta_1 \ldots \delta_n, a\prec e'_1, \ldots, e'_n\succ) \mid \\
&\qquad (\delta_1, e'_1) \in V_\rho(e_1), \ldots, (\delta_n, e'_n) \in V_\rho(e_n)\} \\
V_\rho(\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e) &= \{(D.t_i\delta, e') \mid i \in \{1, \ldots, n\}, (\delta, e') \in V_\rho(\lfloor e \rfloor_{D.i})\}
\end{aligned}$$

Figure 6.11: Computing partial semantics.

6.5.4 Computing the Semantics

The final piece needed to define the denotational semantics of CCC expressions is the function $V_\rho(e)$, which computes the partial semantics of e in the context of environment ρ. The definition of this function is given in Figure 6.11. Because of the groundwork laid in the previous subsections, there should be few surprises.

For the three lambda calculus constructs, the definition is very straightforward. For a variable reference v, it returns the partial semantics bound to v in ρ. For an application, it computes the partial semantics of each subexpression and composes the results as described in Section 6.5.3. For an abstraction, it produces a trivial mapping to a closure. Note that if an unbound variable is encountered, lookup will fail and the semantics is undefined.

There are two cases for object structures. For a leaf, we return the empty decision mapped to the leaf. For an internal node we compute the partial semantics of each subexpression and concatenate *all combinations* of the results. This effectively computes the product of the semantics of the subexpressions.

For a dimension declaration, we select each tag t_i in turn, compute the partial semantics of $\lfloor e \rfloor_{D.i}$, and prepend $D.t_i$ to each decision in the result. This will eliminate all choices bound by D. In the event of an unbound choice, the entire semantics is undefined.

Finally, we can use the function in Figure 6.11 to define the semantics of CCC expressions as $[\![e]\!] = V_\varnothing(e)$, where $\varnothing$ is the empty environment. Note that $[\![e]\!]$ is also undefined whenever $rng(V_\varnothing(e))$ contains a closure since we require $[\![\cdot]\!]$ to map to plain expressions.

6.6 Comparison of Relative Local Expressiveness

We have claimed that the compositional choice calculus subsumes the annotative and compositional approaches to feature implementation and is indeed more powerful than either approach on its own. We have provided example-based evidence of these claims throughout the chapter. In this section, we make these comparisons more formally and directly, using Felleisen's framework for comparing the relative local expressiveness of languages [1991].

Local expressiveness is not the same as computational expressiveness—given two Turing-complete languages L_1 and L_2, it is possible for L_1 to be more *locally expressive* than L_2 if L_1 contains expressions that cannot be locally transformed into *operationally equivalent* expressions in L_2, and the reverse is not also true. Two expressions in different languages are operationally equivalent if they resolve to the same denotations according to the semantics functions of their respective languages.

We compare three languages: the compositional choice calculus (CCC), the annotative choice calculus extended with the **share** construct (ACC) to represent annotative approaches, and a new language, the computational feature algebra (CFA) to represent compositional approaches. We define CFA to be the set of all variation-free (no dimensions or choices) CCC expressions. Thus, CFA is a conservative extension of the AHEAD feature algebra [Batory et al., 2004] (see Section 6.1.1), which consists of only the application and structure constructs of CCC. The additional lambda calculus constructs give CFA metaprogramming capabilities not available in AHEAD's feature algebra, making CFA at least a fair representation of the compositional approach.

Lemma 6.6.1. CCC *is more locally expressive than* CFA.

Proof. CCC is a conservative extension of CFA, by construction. Therefore, we must show that the additional constructs in CCC, dimensions and choices, cannot be locally transformed into operationally equivalent CFA expressions. We do this in several steps.

(1) A CCC choice $D\langle e_1, e_2\rangle$ must be represented in CFA by an application of some function d to e_1 and e_2. Application is the only viable choice of construct here since both e_1 and e_2 must be represented, and it must be possible to reduce the choice to one of these two subexpressions.

(2) Dimension declarations must be represented by an abstraction. In order to resolve the choice $d\ e_1\ e_2$, some selector must be substituted for d. Since d must be scoped and since potentially many choices in the dimension corresponding to d must be synchronized, d must be a lambda-bound variable.

(3) Following from (1) and (2), tag selection must be represented by applying the abstraction binding d to some selector. For example, to select e_1 from the choice bound by dimension d, we can write $(\lambda d.(d\ e_1\ e_2))\ (\lambda x.\lambda y.x)$.

(4) Consider the following CCC expression in which e_1 and e_2 are both variation free.

$$e_{\text{CCC}} = \mathbf{dim}\ D\langle t_1, t_2\rangle\ \mathbf{in}\ D\langle e_1, e_2\rangle$$

Assume that it is possible to locally transform e_{CCC} into an operationally equivalent CFA expression e_{CFA}. Then, given a context C (which we assume without loss of generality is also variation free), $C[e_{\text{CCC}}]$ is operationally equivalent to $C[e_{\text{CFA}}]$. From (1) and (2), e_{CFA} has the following form.

$$e_{\text{CFA}} = \lambda d.(d\ e_1\ e_2)$$

However, $C[e_{\text{CCC}}]$ is *not* operationally equivalent to $C[e_{\text{CFA}}]$ since it violates (3). Specifically, the context C prevents us from applying the abstraction to a selector. The only way to resolve this is by lifting the abstraction out of the context.

$$e_{\text{CFA}}{}' = \lambda d.C[(d\ e_1\ e_2)]$$

Now $e_{\text{CFA}}{}'$ is operationally equivalent to $C[e_{\text{CCC}}]$ but the transformation is non-local, since it escapes the context C. Thus, by contradiction, e_{CCC} cannot be locally transformed into an operationally equivalent expression in CFA. □

Lemma 6.6.2. CCC *is more locally expressive than* ACC*.*

Proof sketch. This case is harder since CCC is *not* a conservative extension of ACC—the **share** construct exists in ACC but not in CCC. Furthermore, the ACC expression $e_{\text{ACC}} = \textbf{share}\ v\ := e\ \textbf{in}\ e'$ is not operationally equivalent to the CCC expression $e_{\text{CCC}} = (\lambda v.e')\ e$, as we might expect, because of staging differences in the languages' semantics. Suppose a dimension D is declared in e and that e' contains $n > 1$ references to v. In ACC, we will make just one selection in D since **share** expressions are expanded only after all dimensions have been eliminated. In CCC, however, β-reduction and dimension elimination are interleaved, so the declaration of D will be multiplied n times when e_{CCC} is reduced, requiring up to n separate selections in D.

To show that there is no loss of expressiveness from ACC to CCC, we must provide a local transformation from e_{ACC} to an operationally equivalent CCC expression. We only describe this transformation at a high level here. The individual steps, however, are just applications of the semantics-preserving transformation laws for ACC expressions, defined in Section 3.5 and Section 4.5, and proved correct in our previous work [2011b]. (Note that we will apply the transformation laws only to the ACC expression, prior to converting it to CCC, so these previous results can be reused in full.)

We begin by observing that if the bound expression e is dimension free, then e_{ACC} and e_{CCC} are already operationally equivalent since no dimensions will be multiplied when e_{CCC} is reduced. Therefore, the transformation consists of two steps. First, we use the transformation laws to transform e_{ACC} into a semantically equivalent ACC expression in which all dimension declarations have been factored out of the bound expression. Second, we replace each **share** expression resulting from this transformation with an abstraction-application pair, completing the transformation to an operationally equivalent CCC expression.

The preconditions of the transformation laws reveal that the first step of the above transformation is potentially complicated by the presence of (1) dependent dimensions in e since dimensions cannot be factored out of their enclosing choices, and (2) free choices in e' since they can be captured when factoring dimensions out of e. Both problems can be resolved by first factoring the offending choices out of the **share** expression. Arbitrary choice factoring is also supported by the transformation laws.

Because there is a local transformation of e_{ACC} into operationally equivalent CCC, and since all other constructs are the same, CCC can *macro express* ACC [Felleisen, 1991]. Observe that the reverse is trivially false since CCC is Turing complete (see below) and ACC is not. Therefore, CCC is more locally expressive than ACC. □

Lemma 6.6.3. CCC *is more locally expressive than* ACC $\cup$ CFA*.*

Proof. This follows directly from Lemma 6.6.1 and Lemma 6.6.2, combined with the observation that there are expressions in CCC that cannot be locally transformed into either ACC or CFA. Such an example can be constructed by combining the examples from the previous proofs. □

In addition to the results above, we observe that: (1) plain expressions exist in both ACC and CFA (ACC $\cap$ CFA $\neq \varnothing$), (2) dimension declarations exist in ACC but not CFA (ACC $-$ CFA $\neq \varnothing$), and (3) lambda abstractions exist in CFA but not ACC (CFA $-$ ACC $\neq \varnothing$). Putting it all together, we can construct the Venn diagram in Figure 6.12, which illustrates the relative local expressiveness

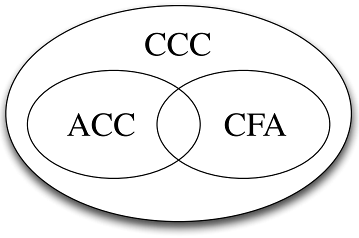


Figure 6.12: Relative local expressiveness of choice calculus variants.

of the three languages. Furthermore, we can observe that both CCC and CFA are Turing complete, since their semantics reduce to the normal-order reduction of lambda calculus terms in the absence of structures, dimensions, and choices.

The compositional choice calculus provides a formal basis for the combination of the compositional and annotative approaches to feature implementation, making it possible to utilize their strengths while mitigating their weaknesses. While we have motivated and introduced CCC from the perspective of FOSD, its intended scope is more general and applies to all kinds of variation representations. The compositional choice calculus is part of our larger goal to explore the potential of *variation programming*, which is concerned with writing programs to generate, query, manipulate, and analyze variation structures. This is the focus of the next chapter.

Chapter 7 – Variational Programming

The semantics of the choice calculus is based on selection, which reduces variability by eliminating choices and dimensions. Although selection is an essential operation, it is only one example of potentially many interesting operations. In this chapter, we will explore the intersection of variation representations and functional programming, which will suggest many more transformations and analyses of variation structures.

The exploration will be supported by a domain-specific embedded language (DSEL) [Hudak, 1998] in Haskell for representing variation, based on the choice calculus. This will allow us to define many sophisticated operations on choice calculus expressions as plain Haskell functions. In Section 7.1, we will present the integration of Haskell and the choice calculus, highlighting the important properties (and some potential problems) of this implementation. This chapter is based on a *Generative and Transformational Techniques in Software Engineering Summer School* tutorial. The accompanying source code is available online.[1] It contains the implementation of the DSEL, all of the examples presented in this chapter, and more.

The integration of the choice calculus with a powerful metalanguage like Haskell provides a way to write programs that query, manipulate, and analyze variation structures. We call this *variational programming*. In Section 7.2, we introduce the basic elements of variational programming by writing functions on variational lists. We illustrate how to generalize standard list functions to work on variational lists, and how to write functions that manipulate choices and dimensions within variational lists. In Section 7.3, we introduce a simplified version of variational Haskell that supports the exploration of operations specific to maintaining variational software.

[1] https://github.com/walkie/CC-GTTSE

This chapter represents the interaction of many different languages. Understanding which languages are involved, what roles they play, and how they are related to one another is important to keep a clear view of the different representations, their purpose, and how variational programming works. Here is a brief summary of the languages involved.

- The *choice calculus* is a generic variation language that can be applied to, or instantiated by, many different *object languages*. Given an object language L, we write $V(L)$ for the result of instantiating the choice calculus with L, as described in Section 3.3.1.
- An *object language*, such as a list data type or Haskell, can be used to represent plain, non-variational terms, such as lists and Haskell programs. An object language can be made variational by instantiating the choice calculus with it.
- A *variational language* is the result of instantiating the choice calculus with an object language. We write VL for the variational version of the object language L, that is, $VL = V(L)$. For example, we can refer variational lists as VList, and variational Haskell programs as VHaskell.
- We also use Haskell as a *metalanguage* to do variational programming. We represent the choice calculus, object languages, and variational languages all as data types in Haskell to facilitate the writing of variational programs.

More extended discussions of the latter three of these terms can be found in Section 2.1.

Finally, in this chapter we assume some basic familiarity with Haskell. Specifically, we assume knowledge of functions and data types and how to represent languages as data types. Knowledge of monads and type classes is useful, but not strictly required.

7.1 A Variation DSEL in Haskell

The choice calculus, as presented in Chapter 3 and Chapter 4 is an entirely static representation. It allows us to precisely specify how a program varies, but we cannot use the choice calculus itself to edit, analyze, or transform a variational program. Throughout this thesis, we have defined operations on the choice calculus using mathematical notation—for example, defining choice elimination to support the definition of the denotational semantics in Section 3.4. In some regards, math is an ideal metalanguage since it is infinitely extensible and extremely flexible—we can define almost any operation we can imagine. However, it is difficult to test an operation defined in math or to apply it to several examples quickly to observe its effect. In other words, it is hard to play around with mathematical definitions. This is unfortunate since playing around can often lead to challenged assumptions, clever insights, and a deeper understanding of the problem at hand.

In this section, we introduce a DSEL for constructing and manipulating variational data structures. This DSEL is based on the choice calculus, but is much more powerful since we have the full expressiveness of our metalanguage (Haskell) at our disposal. Using this DSEL, we can define all sorts of new operations for querying and manipulating variational artifacts. Because the operations are defined in Haskell, certain correctness guarantees are provided by the type system, and most importantly, we can execute the operations and observe the outputs. In this way, the DSEL supports a hands-on, exploratory approach to variation research. It also allows us to explore the interaction of variation representations and strongly typed functional programming.

In the DSEL, all of the relevant languages are represented as Haskell data types. This includes the choice calculus and any object languages we might want to instantiate it with. The representation of the choice calculus in the DSEL is given in Figure 7.1. It adapts the dimension declaration and choice constructs from the choice calculus into Haskell data constructors, `Dim`

```
type Dim = String
type Tag = String

data V a = Obj a
         | Dim Dim [Tag] (V a)
         | Chc Dim [V a]
```

Figure 7.1: The choice calculus as a Haskell data type.

and `Chc`. The `Obj` constructor will be explained below. Dimension and tag names are captured by the `Dim` and `Tag` types, which are just synonyms for the predefined Haskell type `String`.

The `V a` data type serves as the generic representation of variation within our DSEL. The type constructor name `V` is intended to be read as "variational", and the type parameter `a` represents the object language to be varied. So, given a type `Haskell` representing Haskell programs, the type `V Haskell` would represent variational Haskell programs (see Section 7.3).

The `Obj` constructor is roughly equivalent to the object structure construct from the choice calculus. However, here we do not explicitly represent the structure as a tree, but rather insert an object language value directly. An important feature of the DSEL is that it is possible for the data type representing the object language, corresponding to the type parameter `a`, to itself contain variational types (created by applying the `V` type constructor to its argument types), and operations written in the DSEL can query and manipulate these nested variational subexpressions generically. This is achieved through the use of the "scrap your boilerplate" (SYB) library [Lämmel and Peyton Jones, 2003, 2004] which imposes a few constraints on the structure of `a`. These constraints will be described in Section 7.2.1. In the meantime, we will only use the very simple object language of integers, `Int`, which cannot contain nested variational subexpressions.

One of the advantages of using a metalanguage like Haskell is that we can define functional shortcuts for common syntactic forms. In Haskell, these are often called "smart constructors". For example, we define the following function `atomic` for defining atomic dimensions—that is, a dimension with a single choice as an immediate subexpression.

```
atomic :: Dim -> [Tag] -> [V a] -> V a
atomic d ts cs = Dim d ts $ Chc d cs
```

Since we will be presenting a lot of examples throughout this chapter, we also introduce a couple smart constructors to make these examples shorter. First we define a smart constructor for declaring a dimension `A` with tags `a1` and `a2`.

```
dimA :: V a -> V a
dimA = Dim "A" ["a1","a2"]
```

And also a smart constructor for creating choices in this dimension.

```
chcA :: [V a] -> V a
chcA = Chc "A"
```

Both the `dimA` and `chcA` smart constructors are defined by partially applying the constructors `Dim` and `Chc` respectively.

Note that we have omitted the sharing-related constructs introduced in Chapter 4 from the definition of the `V a` data type. This decision was made primarily for two reasons. First, the features provided by the choice calculus **macro** construct are provided by Haskell directly, for example, through Haskell's `let` and `where` constructs. In fact, sharing is much more powerful in Haskell than in the choice calculus since we can also share values via functions. Second, the inclusion of an explicit **share** or **macro** construct greatly complicates some important results later. In particular, we will show that the `V` type constructor is a monad. It is unclear whether this is true when the `V a` data type contains explicit sharing constructs. Several other operations are also more difficult to define with choice calculus-level sharing and macros.

However, there are some subtle implications and drawbacks to relying on Haskell for sharing, compared to the extensions presented in Chapter 4. The first is that it is quite difficult to reproduce the semantics of the **share** construct. Recall that sharing introduced by **share** is expanded only after all dimensions and choices in an expression have been resolved. This allowed us to, for example, write an expression like the following, where only a single selection is made for dimension A, and the result of the choice is reused at both uses of v.

$$\textbf{share}\ v := (\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle 1, 2\rangle)\ \textbf{in}\ v{+}v$$

This example has just two variants: 1+1 and 2+2. However, a similar expression in our DSEL, shown below, has four variants since the `let` expression is conceptually expanded before any subsequent operation on the term.

```
let v = dimA (chcA [Obj 1, Obj 2]) in v+v
```

In fact, there is no easy way to translate a **share** expression that contains dimensions in the bound expression into an equivalent expression in our DSEL. An expression with the same semantics can be achieved by factoring all dimensions out of the bound expression, but this increases the scope of the dimension, potentially causing other problems.

As the previous example demonstrates, Haskell's `let` expression is more similar to the **macro** construct introduced in Chapter 4 since it will duplicate any dimension declarations in its bound expression. However, it is not precisely equivalent. In particular, the lexical scoping of dimensions is not preserved in the DSEL. In the choice calculus, the dimension that binds a choice can always be determined by examining the static context that the choice exists in. For example, in the following choice calculus expression, the choice in A is unbound.

$$\textbf{macro}\ v := A\langle 1, 2\rangle\ \textbf{in dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ v$$

While in the corresponding DSEL expression, the choice in `A` is bound by the dimension surrounding the variable reference. This is demonstrated by evaluating the following DSEL expression (for example, in GHCi), and observing the pretty-printed output.

```
> let v = chcA [Obj 1, Obj 2] in dimA v
dim A<a1,a2> in A<1,2>
```

Once again, there is no way to enforce lexical scoping since the `let` expression is not observable to any operation subsequent operation on the term.

The lack of lexical scoping can lead to the problem of *choice capture* described in Section 4.4. It is not just a problem with `let` expressions, but a fundamental issue related to scoping in DSELs. It is especially serious when a choice intended to be bound by one dimension ends up being bound by another. As an example, consider the following operation `insertA` that declares a dimension `A`, then inserts a choice in `A` into some expression, according to the argument function.

```
insertA :: (V Int -> V Int) -> V Int
insertA f = dimA (f (chcA [Obj 1, Obj 2]))
```

The author of this operation expects the inserted choice to be bound by the dimension declared in this definition, but if the argument function also declares a dimension `A`, the choice could be captured, as demonstrated below.

```
> insertA (\v -> Dim "A" ["a3","a4"] v)
dim A<a1,a2> in dim A<a3,a4> in A<1,2>
```

Now the choice is bound by the dimension in the argument, rather than the intended dimension declared in the `insertA` function.

One way to avoid the problem of choice capture in our DSEL would be to reuse the naming and scoping mechanisms of Haskell to implement the declaration and scoping of choice calculus dimensions. This is a DSEL implementation technique known as *higher-order abstract syntax* [Miller, 2000]. However, this ends up losing the choice and dimension structure of variational expressions entirely, and so does not serve our primary goal of exploring operations on this structure. The compositional choice calculus from Chapter 6 does not exhibit this problem since we have complete control over the evaluation semantics. In CCC we can define the `insertA` function as $\lambda f.\mathbf{dim}\ A\langle a_1, a_2\rangle\ \mathbf{in}\ f\ A\langle 1, 2\rangle$ and the choice in A will always be bound to the declaration that lexically encloses it. Since we do not have the same amount of control over Haskell, however, choice capture is a problem that we must simply acknowledge and cope with.

An important quality of the `V` type constructor, and one of the main motivations for excluding explicit sharing constructs, is that it is both a functor and a monad. Functors and monads are two of the most commonly used abstractions in Haskell. By making the variation representation an instance of Haskell's `Functor` and `Monad` type classes, we make a huge body of existing functions and knowledge instantly available from within our DSEL, greatly extending its syntax. Functors are simpler than (and indeed a subset of) monads, so we will present the `Functor` instance first below. The `Functor` type class contain one method, `fmap`, for mapping a function over a data structure while preserving its structure.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

For `V`, this operation consists of applying the mapped function `f` to the values stored at `Obj` nodes, and propagating the calls into the subexpressions of `Dim` and `Chc` nodes.

```
instance Functor V where
  fmap f (Obj a)      = Obj (f a)
  fmap f (Dim d ts v) = Dim d ts (fmap f v)
  fmap f (Chc d vs)   = Chc d (map (fmap f) vs)
```

Consider the following variational integer expression `ab`, where `dimB` and `chcB` are smart constructors similar to `dimA` and `chcA`.

```
> let ab = dimA $ chcA [dimB $ chcB [Obj 1, Obj 2], Obj 3]
> ab
dim A<a1,a2> in A<dim B<b1,b2> in B<1,2>,3>
```

Using `fmap`, we can, for example, increment every object value in a variational integer expression.

```
> fmap (+1) ab
dim A<a1,a2> in A<dim B<b1,b2> in B<2,3>,4>
```

Or we can map the function `odd :: Int -> Bool` over the structure, producing a variational boolean value of type `V Bool`.

```
> fmap odd ab
dim A<a1,a2> in A<dim B<b1,b2> in B<True,False>,True>
```

The definition of the `Monad` instance for `V` is similarly straightforward. The `Monad` type class requires the implementation of two methods: `return` for injecting a value into the monadic type, and `>>=` (pronounced "bind") for sequentially composing a monadic value with a function that produces another monadic value.

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

```
instance Monad V where
  return = Obj
  Obj a     >>= f = f a
  Dim d t v >>= f = Dim d t (v >>= f)
  Chc d vs  >>= f = Chc d (map (>>= f) vs)
```

Figure 7.2: Monad instance for the `V` type constructor.

The monad instance for `V` is given in Figure 7.2. The `return` method is trivially implemented by the `Obj` data constructor. For the `>>=` method, at an `Obj` node, we simply return the result of applying the function to the value stored at that node. For dimensions and choices, we must again propagate the bind downward into subexpressions.

The effect of a monadic bind is to replace every value in the structure with another monadic value (of a potentially different type) and then to flatten the result. The `concatMap` function on lists is a classic example of this pattern (though the order of arguments is reversed). In the context of variation representations, we can use this operation to introduce new variability into an expression. For example, consider again the expression `ab`. We can add a new dimension `S`, indicating whether or not we want to square each value (the line break in the output was inserted manually).

```
> let chcS i = Chc "S" [Obj i, Obj (i*i)]
> Dim "S" ["n","y"] (ab >>= chcS)
dim S<y,n> in dim A<a1,a2> in
A<dim B<b1,b2> in B<S<1,1>,S<2,4>>,S<3,9>>
```

Each value in the original expression `ab` is expanded into a choice in dimension `S`. The resulting expression remains of type `V Int`. If we had instead used `fmap` instead of `>>=`, the result would have been of type `V (V Int)`, illustrating the flattening quality of monadic bind.

Finally, the DSEL provides several functions for analyzing variational expressions. For example, the function `freeDims :: V a -> Set Dim` returns the set of all free dimensions in a given variational, that is, the dimensions of all choices not bound by a corresponding dimension declaration. Several other basic static analyses are also provided. Most importantly, a semantics function for variational expressions, `sem`, is provided. This is based on the semantics of the choice calculus presented in Section 3.4. Similarly, the semantics of a variational expression of type `V a` is a mapping from decisions (lists of qualified tags) to plain expressions of type `a`. More commonly, we use a function `psem` which computes the semantics of an expression and pretty prints the results. For example, the pretty printed semantics of the expression `ab` is shown below.

```
> psem ab
[A.a1,B.b1]  =>  1
[A.a1,B.b2]  =>  2
[A.a2]  =>  3
```

Each entry in the semantics is shown on a separate line, with a decision on the left of each arrow and the resulting plain expression on the right.

While this section provided a brief introduction to some of the features provided by the DSEL, the following sections on variational programming will introduce many more. In particular, Section 7.2.1 will describe how to make a non-trivial data type variational, Section 7.2.2 and Section 7.2.3 will present a subset of the language designed for the creation of complex editing operations on variational expressions.

7.2 Variational Lists

We start exploring the notion of variational programming with lists, which are a simple but expressive and pervasive data structure. The familiarity with lists will help us to identify important patterns when we generalize traditional list functions to the case of variational lists. The focus on a simple data structure will also help us point out the added potential for variational programming. We present variational programming with lists in several steps.

First, we explain the data type definition for variational lists and present several examples together with some helper functions in Section 7.2.1. Second, we develop variational versions for a number of traditional list functions in Section 7.2.2. In doing this we observe that, depending on the types involved, certain patterns of recursion become apparent. Specifically, we will see that depending on the role of variation in the types of the defined functions, variational parts have to be processed using `fmap`, effectively treating them in a functorial style, or using `>>=`, treating them as monadic values.

In Section 7.2.3 we turn our attention to editing operations for variational lists. While the adapted standard list functions will naturally produce variational results (such as lists, numbers, etc.), this variation is incidental. That is, the results are variational because the arguments passed in were variational, but no new variation is introduced. In contrast, list editing operations introduce or change the variation structure purposefully. We will present in Section 7.2.4 some comments and observations on the different programming styles employed in Section 7.2.2 and Section 7.2.3.

As a motivating example consider how to represent menu preferences using choices and dimensions. Suppose that we prefer to order meat or pasta as the main course in a restaurant and that with meat we always order French fries on the side. Also, if we order pasta, we may have cake for dessert. Using the choice calculus we can represent these menu options as follows (here ϵ represents an empty token that, when selected, does not appear in the list as an element but rather disappears).

$$\begin{array}{l}\textbf{dim}\ \mathit{Main}\langle \mathit{meat}, \mathit{pasta}\rangle\ \textbf{in}\\ \mathit{Main}\langle \texttt{[Steak,Fries]}, \texttt{[Pasta,}\textbf{dim}\ \mathit{Dessert}\langle \mathit{yes}, \mathit{no}\rangle\ \textbf{in}\ \mathit{Dessert}\langle \texttt{Cake}, \epsilon\rangle\texttt{]}\rangle\end{array}$$

Here we have used a simple list notation as an object language. This notation leaves open many questions, such as how to nest lists and how to compose a variational list and a list without variations. We will look at these questions in more detail in the following.

7.2.1 Representing Variational Lists

Lists are typically represented using two constructors: one for an empty list and one for adding a single element to a list. Since lists are the most important data structures in functional programming, they are predefined in Haskell and supported through special syntax. While this is nice, it prevents us from modifying the representation in order to define variational lists. Therefore, we have to define our own list representation first, which we then can extend in a variety of ways to discus the transition to variational lists. A standard definition of a list data type is given below.

```
data List a = Cons a (List a)
            | Empty
```

To create variational lists using the `V a` data type, we have to apply the `V` type constructor somewhere in this definition. One possibility is to apply `V` to `a` thus making the elements in a list variable.

```
data List a = Cons (V a) (List a)
            | Empty
```

While this definition allows us to vary the elements within a list, it does not allow us to vary lists themselves. For example, we *cannot* represent a list whose first element is `1` and whose tail is a either `[2]` or `[3,4]`. Moreover, if this definition were sufficient, we would could use it with the built-in Haskell lists since it defines a data type equivalent to the type `[V a]`.

The limitation described above results from the fact that we cannot have a choice in the second argument of `Cons`. This shortcoming can be addressed by adding another application of `V` to the tail.

```
data List a = Cons (V a) (V (List a))
            | Empty
```

This representation avoids the above problem and is indeed the most general representation imaginable. However, the implementation of our DSEL requires that `V` not be applied to different types within the same data type. This is a technical limitation related to our use of the SYB library.[2] Fortunately, as the eventual solution will show, this feature is not needed for variational lists or for many other interesting applications of variational programming.

A drawback of either of the two previous approaches is that changing the types of constructors can break existing code. This drawback is significant when variation is added post hoc to existing data structures. In such situations we would like to be able to continue using existing functions with minimal changes to existing code.

Therefore, we choose the following representation in which we simply add a new constructor, which serves as a hook for introducing any kind of variation. This satisfies the constraint that `V` be applied to only one type within the `List` data type and is also maximally expressive.

```
data List a = Cons a (List a)
            | Empty
            | VList (VList a)
```

This definition yields what we call an *expanded list*, where "expanded" means that it can contain variational data. However, this expansion is not enough,

[2]It is possible to lift this constraint, but doing so requires quite complex generic programming techniques that make the language more difficult to use. A maximally generic implementation, illustrating how the constraint can be avoided, is available online at `https://github.com/walkie/CC-Generic`.

we also need a type for *variational lists*, that is, lists that are the object of the V type constructor. We introduce a type abbreviation for this type.

```
type VList a = V (List a)
```

The two types List a and VList a for expanded and variational lists, respectively, depend mutually on one another. Through this recursion they accomplish the lifting of the plain list data type into its fully variational version. Note that we use the convention of using the same name for the additional constructor as for the variational type, in this case VList. This helps to keep the variational code more organized, in particular, in situations where multiple variational data types are used.

With the representation of variational lists settled, let us return to the menu example from earlier in this section and see how we can represent it within the DSEL. First, we introduce a data type for representing the various food items involved.

```
data Food = Steak | Pasta | Fries | Cake
```

Although it is not shown, note that in the Haskell definition of this data type we derive instances for the Eq, Show, Data, and Typeable type classes. Instances of Data and Typeable are required for the SYB library to work, while Eq and Show instances are derived for the usual reasons. Unless stated otherwise, every data type used throughout this chapter also derives instances for these classes.

We also introduce several smart constructors and auxiliary functions to make writing variational lists more concise. These are listed in Figure 7.3. The functions single and many build expanded lists from a single element or a plain Haskell list of elements, respectively. The function list lifts an expanded list to a variational list—we will return to this simple definition in a bit. The value vempty represents an empty variational list, vsingle constructs a variational list containing one element, and vcons adds an element to

```
                                  vempty :: VList a
                                  vempty = list Empty

list :: List a -> VList a         vsingle :: a -> VList a
list = Obj                        vsingle = list . single

single :: a -> List a             vcons :: a -> VList a -> VList a
single a = Cons a Empty           vcons x = list . Cons x . VList

many :: [a] -> List a             vlist :: [a] -> VList a
many = foldr Cons Empty           vlist = list . many
```

Figure 7.3: Smart constructors for expanded and variational lists.

the beginning of a variational list. Finally, the function `vlist` transforms a regular Haskell list into a `VList`, which lets us reuse Haskell list notation in constructing variational lists.

Although the `list` smart constructor is just a synonym for the `Obj` data constructor, this trivial definition serves two useful purposes. First, it has a more constrained type than `Obj`—the argument to `list` must be an expanded list, whereas `Obj` accepts arguments of any type. This is useful since, for example, it can help improve error location in the event of a type error. Second, it is more evocative in the sense that it explicitly indicates that a list is being lifted to the variational level, making functions that manipulate variational lists more readable. We use similar synonyms for other object languages (for example, `int` or `haskell`, for the object languages of integers and Haskell code). For consistency of capitalization, we also use the synonym `obj` for generic values.

Now we can define our menu preferences example as a variational list within our DSEL; this is shown in Figure 7.4. A value of type `Menu` is a variational list of food items, while the term `menu` represents our specific

```
type Menu = VList Food

dessert :: Menu
dessert = atomic "Dessert" ["yes","no"] [vsingle Cake,vempty]

menu :: Menu
menu = atomic "Main" ["meat","pasta"]
              [vlist [Steak,Fries],Pasta ‘vcons‘ dessert]
```

Figure 7.4: Menu preferences as a variational list.

menu preferences. We can examine the variational expression we have built by evaluating `menu`, which will pretty print the variational list (the line breaks were inserted manually).

```
> menu
dim Main<meat,pasta> in
    Main<[Steak;Fries],
         [Pasta;dim Dessert<yes,no> in Dessert<[Cake],[]>]>
```

Note that we have defined the pretty printing for the `List` data type to be similar to ordinary lists, except that we use `;` to separate list elements. In this way we keep a notation that is well established but also provides visual cues to differentiate between plain Haskell lists and expanded/variational lists in our DSEL.

We can print the semantics of `menu` in order to enumerate all of the possible menus and the selections necessary to produce them.

```
> psem menu
[Main.meat]  =>  [Steak;Fries]
[Main.pasta,Dessert.yes]  =>  [Pasta;Cake]
[Main.pasta,Dessert.no]  =>  [Pasta]
```

Recall that dependent dimensions are only relevant if the tags they are dependent on are also selected. In this case, we only choose whether to have dessert if we chose pasta for a main course.

Before we move on to discuss variational list programs, we show a couple of operations to facilitate a more structured construction of variation lists. These operations are not very interesting from a transformational point of view, but they can be helpful in decomposing the construction of complicated variational expressions into an orderly sequence of steps.

In the construction of `menu` we can identify two patterns that warrant support by specialized operations. First, in the definition of `dessert` we introduced a dimension representing something that is *optional*. Since this is a common use case, we define a smart constructor `opt` for introducing an optional feature in a straightforward way.

```
opt :: Dim -> a -> VList a
opt d x = atomic d ["yes","no"] [vsingle x,vempty]
```

Second, the structure of dimension declarations forces us to separate tags and the elements they label, even for atomic dimensions. A more modular definition of the `menu` example can be given if we define the different menu options separately, then combine them to form menu. This idea is quite similar to the "direct tagging" approach to representing choices, discussed in Section 3.2.2. The only difference is that we will represent tagged alternatives separately, then combine them later to form a choice. To support this approach, we first introduce a simple representation of tagged alternatives as a pair of a tag and a variational value.

```
type Tagged a = (Tag, V a)
```

Then we introduce some syntactic sugar for defining tagged alternatives, in the form of an infix operator `<:`.

```
dessert = opt "Dessert" Cake
meat    = "meat"  <: vlist [Steak,Fries]
pasta   = "pasta" <: Pasta ‘vcons‘ dessert
menu    = alt "Main" [meat,pasta]
```

Figure 7.5: Alternative definition of `menu` with tagged alternatives.

```
infixl 2 <:
(<:) :: Tag -> V a -> Tagged a
t <: v = (t,v)
```

Finally, we can define an operation `alt` that combines a list of tagged alternatives into the declaration of an atomic dimension.

```
alt :: Dim -> [Tagged a] -> V a
alt d tvs = atomic d ts vs where (ts,vs) = unzip tvs
```

Putting this all together, we can redefine our `menu` example using tagged alternatives as shown in Figure 7.5. This definition produces a syntactically identical variational list as our original definition above, but we have achieved the ability to separate each course into its own definition. This makes it easier to define new kinds of menus by combining existing courses in new ways.

7.2.2 Standard Variational List Functions

In this subsection we will illustrate how to implement for variational lists some of the standard functions for transforming and aggregating lists. We will do this at first through direct pattern matching and recursion. Later, we will introduce more general variational list operations, such as map and fold.

Let us start by implementing the function `len` to compute the length of a variational list. The definition of this functions is given in Figure 7.6, which we'll analyze in depth. The first thing to notice is that the return type of the function is not just `Int` but rather `V Int` since a variational list may represent plain list variants of different lengths. The implementation of the `leng` function proceeds by pattern matching. There are three cases to consider for the `Empty`, `Cons`, and `VList` constructors.

- The length of the empty list is zero. However, we must be careful to not just return `0` since the return type of the function must be `V Int`. Therefore we lift the plain integer `0` into a variational integer using the `int` function. This is just a type synonym for `Obj`, as motivated in the discussion of the function `list` in the previous subsection. Note that we could also have used the method `return` from the `Monad` type class.

- For the `Cons` case, the length of a non-empty list is given by the length of the tail plus one. Again, because of the variational return type of `len` we cannot simply add one to the result of the recursive call. Since `len xs` can produce, in general, an arbitrarily complex variation expression overs integers, we have to make sure to add one to all variants. This can be accomplished by the functor method `fmap`.

- For the `VList` case, we must compute the length of a variational list (of type `V List`) directly. We do this by mapping the `len` function across the variational list and flattening the result, which corresponds to the monadic bind observation. We might initially think of using `fmap` again here, but examining the types reveals that this is not the right approach since it would lead to a result of type `V (V Int)`, when we actually want a result of type `V Int`.

Note that in each of the above cases, the correct implementation was motivated primarily by the types involved.

Returning to our menu example from the previous subsection, observe that we cannot apply `len` directly to `menu` since the types do not match up—`len` expects an argument of type `List a`, but `menu` has type `VList Food`. Therefore, it seems we need an additional length function that can be applied

```
len :: List a -> V Int
len Empty       = int 0
len (Cons _ xs) = fmap (+1) (len xs)
len (VList vl)  = vl >>= len
```

Figure 7.6: Computing the variational length of an expanded list.

to values of type `VList a`. In fact, we have defined such a function already in the `VList` case of `len`, so we can simply reuse that definition.

```
vlen :: VList a -> V Int
vlen vl = vl >>= len
```

However, it turns out that we need to perform such a lifting of an argument to a variational type quite often, so we define a general function for that purpose. The function `liftV` takes a function from some type `a` to a variational type `V b` and returns a function of type `V a -> V b`, where the argument has been made variational.

```
liftV :: (a -> V b) -> V a -> V b
liftV = flip (>>=)
```

With `liftV` we can redefine `vlen` equivalently as follows.

```
vlen :: VList a -> V Int
vlen = liftV len
```

We generally use this naming convention for functions. Given a function `f` whose input is of type `T` we use the name `vf` for its lifted version that accepts arguments of type `V T`.

We can now test the definition of `vlen` by applying it to the variational list `menu` defined in Section 7.2.1.

```
> vlen menu
dim Main<meat,pasta> in
Main<2,dim Dessert<yes,no> in Dessert<2,1>>
```

As expected the result is a variational expression over integers. It represents the number of food items we have depending on our decisions about the menu. Computing the semantics of this expression makes the mapping from decisions to number of food items more obvious.

```
> psem $ vlen menu
[Main.meat]  =>  2
[Main.pasta,Dessert.yes]  =>  2
[Main.pasta,Dessert.no]  =>  1
```

Comparing this output to the semantics of `menu` in the previous subsection, it is easy to confirm that the result is correct.

We have explained the definition of `len` in some detail to illustrate the considerations that led to the implementation. We have tried to emphasize that the generalization of a function definition for ordinary lists to variational lists requires mostly a rigorous consideration of the types involved. In other words, making existing implementations work for variational data structures is an exercise in *type-directed programming* in which the types dictate (to a large degree) the code [Wadler, 1989].

We examine one other basic list operation in detail before moving on to more general functions on variational lists. This will highlight an important pattern in the generalization of list functions to the variational case.

A function `cat` for concatenating expanded lists is given in Figure 7.7. The cases for `Empty` and `Cons` are easy and follow the definition for ordinary lists—in the case of `Empty`, return the second list, for `Cons`, recursively append the second list to the tail of the first. However, the definition for a variational list is not so obvious. If the first list is given by a variational expression `vl`, we have to make sure that we append the second list to *all* of the lists that

```
cat :: List a -> List a -> List a
cat Empty      r = r
cat (Cons a l) r = Cons a (l ‘cat‘ r)
cat (VList vl) r = VList (fmap (‘cat‘ r) vl)
```

Figure 7.7: Concatenating expanded lists.

are represented by `vl`. We have seen that we have, in principle, two ways to do that, namely `fmap` and `>>=`. Again, a close look at the involved types will tell us what the correct choice is. First we observe that the result of `cat` is the same as the type of its arguments, `List a`. Therefore we can simply apply the function `cat` at every `List a` nested within `vl`, and we do not need to do any subsequent flattening. This corresponds to the function `fmap`. The situation for `len` was different because its result was a variational type, which required flattening the newly created `V` structures with `>>=`.

As with `len`, we also need a version of `cat` that works for variational lists, and not just expanded lists. A simple solution is to inject the variational list arguments into the `List` type using the `VList` constructor, facilitating a direct application of `cat`.

```
vcat :: VList a -> VList a -> VList a
vcat l r = list $ cat (VList l) (VList r)
```

To show `vcat` in action, assume that we extend `Food` by another constructor `Sherry` which we use to define the following variational list representing a potential drink before the meal.

```
aperitif :: VList Food
aperitif = opt "Drink" Sherry
```

When we concatenate the two variational lists `aperitif` (with two variants) and `menu` (with three variants), we obtain a variational list that contains a total of six variant lists.

```
nth :: Int -> List a -> V a
nth _ Empty        = undefined
nth 1 (Cons x _)   = obj x
nth n (Cons _ xs) = nth (n-1) xs
nth n (VList vl)   = vl >>= nth n
```

Figure 7.8: Indexing into an expanded list.

```
psem $ vcat aperitif menu
[Drink.yes,Main.meat]  =>  [Sherry;Steak;Fries]
[Drink.yes,Main.pasta,Dessert.yes]  =>  [Sherry;Pasta;Cake]
[Drink.yes,Main.pasta,Dessert.no]  =>  [Sherry;Pasta]
[Drink.no,Main.meat]  =>  [Steak;Fries]
[Drink.no,Main.pasta,Dessert.yes]  =>  [Pasta;Cake]
[Drink.no,Main.pasta,Dessert.no]  =>  [Pasta]
```

Since the evaluation of `vcat` duplicates the dimensions in `menu`, the resulting term structure becomes quite difficult to read and understand. Therefore we show only the semantics of the result.

All of the functions we have considered so far have only lists as arguments. Of course, programming with variational lists should integrate smoothly with other, non-variational types. Figure 7.8 defines an indexing function `nth` to compute the nth element of a variational list. Observe that although the integer argument passed to `nth` is not variational, the result `V a` is variational, since the element at position `n` may be different in each variant list.

As usual, we must lift `nth` to work on variational lists. We can reuse the `liftV` function for this.

```
vnth ::  Int -> VList a -> V a
vnth = liftV . nth
```

Note that the index remains non-variational through the function composition—once again the desired type motivates this definition.

```
fold :: (a -> b -> b) -> b -> List a -> V b
fold _ b Empty      = obj b
fold f b (Cons a l) = fmap (f a) (fold f b l)
fold f b (VList vl) = vl >>= fold f b
```

Figure 7.9: Right fold on an expanded list.

As a final example of how to generalize standard list functions to variational lists, and as an example of generalizing a higher-order function, consider the ubiquitous list fold operation (also known as “reduce”). The definition of a right fold operation on expanded lists is given in Figure 7.9. Structurally, it is identical to the `len` operation, where the aggregating function has been abstracted from `+` to `f`. With this in mind, we can redefine `len` in terms of `fold` as follows.

```
len :: List a -> V Int
len = fold (\_ s -> succ s) 0
```

Many more functions on variational lists can be found in the source code accompanying this chapter.

7.2.3 Edit Operations for Variational Lists

The `menu` example that we introduced in Section 7.2.1 was built in a rather ad hoc fashion in one big step from scratch. More realistically, variational structures develop over time, by dynamically adding and removing dimensions and choices in an expression, or by extending or shrinking choices or dimensions. Additionally, the semantics-preserving transformation laws for choice calculus expressions (see Chapter 3) suggest a number of operations for refactoring variational expressions, such as factoring choices or hoisting dimensions.

In this section we will present several operations that can be used to incrementally build and evolve variational lists. Most of these operations are generic in the sense that they can also be applied to other kinds of variational structures. We will demonstrate this by reusing some of them in Section 7.3.

As a motivating example suppose that in our menu example we want to decide first about the dessert and not about the main course. In other words, we want to obtain an alternative representation of our menu with the declaration of the `Dessert` dimension at the top, rather than the declaration of `Main`. Since we don't want to rewrite our menu from scratch, we want some way to hoist the `Dessert` dimension to the top without changing the variants that the menu represents.

We can break down a generalized version of this operation into the following steps. Assume that `e` is the expression to be refactored and `d` is the name of the dimension declaration that is to be moved.

1. Find the declaration of the dimension `d` that is to be moved.
2. If the first step is successful, cut out the found dimension declaration `Dim d ts e'` and remember its context `c`, that is, an expression with a hole. For now, assume `c` is a simple function that takes an expression of the type required at the hole, and produces an expression of the type of the overall expression.
3. Keep the scope of the found dimension declaration, `e'`, at its old location, which can be achieved by applying `c` to `e'`.
4. Finally, move the dimension declaration to the top of the expression, which is achieved by wrapping it around the already modified expression obtained in the previous step; that is, we produce the expression `Dim d ts (c e')`.

To implement these steps we need to solve some technically challenging problems. For example, finding a subexpression in an arbitrary data type expression, removing it, and replacing it with some other expression requires some advanced generic programming techniques.

To support these generic transformations, we employ the SYB [Lämmel and Peyton Jones, 2003, 2004] and the "scrap your zipper" (SYZ) [Adams, 2010] libraries for Haskell. Since a detailed explanation of these libraries is beyond the scope of this thesis, we only briefly describe the most relevant functions as we encounter them.

At a high level, the approach is based on a type `C a`, which represents a context in a type `V a`. Essentially, a value of type `C a` represents a pointer to a subexpression (the hole) within a value of type `V a`, allowing us to extract the subexpression and also replace it.

A context is typically the result of an operation to search for (or "locate") a subexpression within a variational expression. We introduce a type synonym `Locator` for such functions.

```
type Locator a = V a -> Maybe (C a)
```

The resulting `Maybe` type indicates that a search for a context may fail.

Our search for a particular subexpression is typically based on finding a subexpression that has some property, indicated by satisfying a predicate. We introduce another type synonym `Pred` for representing predicates on variational expressions.

```
type Pred a = V a -> Bool
```

Using these types, we introduce the type of a function `find` that performs a preorder traversal of an arbitrary variational expression, locating the topmost, leftmost subexpressions that satisfies the given predicate.

```
find :: Data a => Pred a -> Locator a
```

We omit the implementation of `find` since it relies on details of the SYZ library; this is also the reason for the `Data` type class constraint on `a`.

The function find realizes the first step of the transformation sequence needed for hoisting the Dessert dimension to the top of the variational list menu. menu. All we need is a predicate to identify a particular dimension d, which is straightforward to define.

```
dimDef :: Dim -> Pred a
dimDef d (Dim d' _ _) = d == d'
dimDef _ _             = False
```

Now we can find the declaration of the Dessert dimension in menu by applying find (dimDef "Dessert") menu.

The second step of cutting out the dimension declaration is realized by the function extract, which returns as a result a pair consisting of the context and the subexpression at that context.

```
extract :: Data a => Pred a -> Splitter a
extract p e = do c <- find p e
                 h <- getHole c
                 return (c,h)
```

The function extract is an example of a class of functions that split an expression into two parts, a context plus some additional information about the expression in the hole. In the case of extract that information is simply the expression itself. We introduce a type synonym Splitter to represent the type of such functions.

```
type Splitter a = V a -> Maybe (C a, V a)
```

The definition of extract uses find to locate the context and then simply extracts the subexpression stored in the context using the SYZ function getHole.

The third step of applying the context to the scope of the dimension expression requires the function <@, whose definition is based on SYZ functions that we don't show here.

```
hoist :: Data a => Dim -> V a -> V a
hoist d e = withFallback e $ do
    (c,Dim _ ts e') <- extract (dimDef d) e
    return (Dim d ts (c <@ e'))
```

Figure 7.10: Dimension hoisting operation (unsafe).

```
(<@) :: Data a => C a -> V a -> V a
```

The function `<@` can be understood as simply replacing the expression at the hole specified by the first argument with the expression passed as the second argument.

Finally, we can combine all of these functions to define a function `hoist` in Figure 7.10 for hoisting dimension declarations. Note that the function `withFallback` is a synonym for `fromMaybe`, which we use to return the original expression as a default if the lifting process fails.

Now we can apply `hoist` to `menu` to lift the `Dessert` dimension to the top of the choice calculus expression, producing the expected result. (We save the new menu as a `dMenu` for future reference.)

```
> let dMenu = hoist "Dessert" menu
> dMenu
dim Dessert<yes,no> in dim Main<meat,pasta> in
    Main<[Steak;Fries],[Pasta;Dessert<[Cake],[]>]>
```

There are two obvious shortcomings of the current definition of `hoist`. One problem is that moving the dimension might capture free `Dessert` choices.[3] The other problem is that the decision in the `Dessert` dimension might be made for nothing since it does not have an effect when the next decision in the `Main` dimension is to select `meat`.

[3]Capturing free desserts actually sounds appealing from an application point of view. :-)

```
safeHoist :: Data a => Dim -> V a -> V a
safeHoist d e = withFallback e $ do
    (c,Dim _ ts e’) <- extract (dimDef d) e
    if d ‘Set.member‘ freeDims e
      then Nothing
      else return (Dim d ts (c <@ e’))
```

Figure 7.11: Capture-avoiding dimension hoisting operation.

The first problem can be easily addressed by extending the definition of `hoist` by a check for capturing unbound `Dessert` choices that returns failure (that is, `Nothing`) if `d` occurs as a free dimension in `e`. This failure will be caught eventually by the `withFallback` function that will ensure that the original expression `e` is returned instead. A definition of this "safe" version of hoist is given in Figure 7.11. Recall from Section 7.1 that the function `freeDims` returns a set (as Haskell's built-in `Data.Set` type) of the dimension names of unbound choices.

The second problem with the definition of `hoist` can be observed by comparing the semantics of `dMenu` with the original semantics of `menu` (shown in Section 7.2.1).

```
> psem $ dMenu
[Dessert.yes,Main.meat]  =>  [Steak;Fries]
[Dessert.yes,Main.pasta]  =>  [Pasta;Cake]
[Dessert.no,Main.meat]  =>  [Steak;Fries]
[Dessert.no,Main.pasta]  =>  [Pasta]
```

It is clear that the decision in the `Dessert` dimension has no effect if the decision in the `Main` dimension is `meat`. The reason for this is that the `Dessert` choice appears only in the `pasta` choice of the `Main` dimension. We can fix this by pushing the `Main` choice plus its dimension declaration into the `no` alternative of the `Dessert` choice.

```
prioritize :: Data a => Dim -> Dim -> V a -> V a
prioritize b a e = withFallback e $ do
  (dA,ae)           <- extract (dimDef a) e
  (cA,Chc _ [a1,a2]) <- extract (chcIn a) ae
  (cB,Chc _ [b1,b2]) <- extract (chcIn b) a2
  return $ dA <@ (Chc b [cB <@ b1,cA <@ (Chc a [a1,cB <@ b2])])
```

Figure 7.12: Transformation to eliminate a class of redundant decisions.

This modification is an instance of the following slightly more general transformation schema, which applies in situations where we have two top-level dimension declarations of A and B, but a choice in dimension B is contained in just one alternative of all choices in A. (For simplicity, we show only the special case where A has one choice with two alternatives.)

$$\begin{aligned} &\textbf{dim}\ B\langle b_1, b_2\rangle\ \textbf{in dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;}B\langle\texttt{b1},\texttt{b2}\rangle\texttt{]}\rangle \\ \rightsquigarrow\ &\textbf{dim}\ B\langle b_1, b_2\rangle\ \textbf{in}\ B\langle \texttt{[a2;b1]}, \textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;b2]}\rangle\rangle \end{aligned}$$

The first expression is transformed so that the selection of b_1 is guaranteed to have an effect. That is, when b_1 is selected, we effectively trigger the selection of a_2 by copying the alternative `a2` without the surrounding dimension and choice. This transformation makes the most sense in the case when B represents an optional dimension, that is, $b_1 = yes$, $b_2 = no$, and $\texttt{b2} = \epsilon$, because in this case the selection of $b_2 = no$ makes no difference, regardless of whether we choose a_1 or a_2.

This transformation can be easily extended to the case where dimension A has more than two tags and more than one choice. This requires that each choice in A contains the choice in B in the same alternative.

In Figure 7.12 we define a function `prioritize` that can perform the required transformation automatically. The argument dimension `b` corresponds to B above, and `a` corresponds to A. For simplicity we assume that the choice in b to be prioritized is contained in the second alternative of the choice in a. The function's structure is typical of the generic transformations we can write with the DSEL; it works as follows. First it decomposes the expression to be transformed into a collection of nested contexts and expressions, then it uses these to rebuild the result expression. Specifically, we find the location of the dimension definition for a, remembering it as context dA. Then we find the context cA of the a choice, and the choice to be prioritized in the second alternative of the a choice, a2. Finally, having isolated all of the required subexpressions, we can assemble the result by applying the contexts following the RHS of the above transformation schema.

The contexts of the two choices are found using the extract function with a predicate chcIn that finds a particular choice. It's definition is given below.

```
chcIn :: Dim -> Pred a
chcIn d (Chc d' _) = d == d'
chcIn _ _          = False
```

This is very similar to the dimDef predicate for finding a particular dimension declaration.

Note that the transformation prioritize *does not* preserve the semantics; in fact, the reason for applying it is to make the semantics more compact. However, the transformation is *variant preserving* in the sense described in Section 3.6; that is, the range of the semantics is unchanged, only the descisions needed to generate each variant have changed. This can be observed by comparing the semantics of dMenu shown above with the semantics of dMenu with the Dessert choice prioritized over the Main choice.

```
> psem $ prioritize "Dessert" "Main" dMenu
[Dessert.yes]  =>  [Pasta;Cake]
[Dessert.no,Main.meat]  =>  [Steak;Fries]
[Dessert.no,Main.pasta]  =>  [Pasta]
```

The prioritization of the `Dessert` choice has removed the decision about whether to have meat if we choose `yes` for `Dessert` since we can only have pasta if we choose to have dessert.

As a final example we illustrate how to combine the two transformations we have defined so far, `hoist` and `prioritize`. In terms of the choice calculus, combining dimension hoisting and dimension prioritization leads to a transformation that we call *dependency inversion*.

$$\begin{aligned} &\textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;}\textbf{dim}\ B\langle b_1, b_2\rangle\ \textbf{in}\ B\langle \texttt{b1}, \texttt{b2}\rangle\texttt{]}\rangle \\ \rightsquigarrow\ &\textbf{dim}\ B\langle b_1, b_2\rangle\ \textbf{in}\ B\langle \texttt{[a2;b1]}, \textbf{dim}\ A\langle a_1, a_2\rangle\ \textbf{in}\ A\langle \texttt{[a1]}, \texttt{[a2;b2]}\rangle\rangle \end{aligned}$$

Reusing `hoist` and `prioritize`, the definition of inversion is straightforward.

```
invert :: Data a => Dim -> Dim -> V a -> V a
invert b a = prioritize b a . hoist b
```

The definition of `invert` demonstrates that we can build more complicated variational programs out of simpler components, illustrating the compositional nature of our variation DSEL.

7.2.4 Variational Programming Modes

To close this section, we share a few thoughts on the nature of variational programming. Section 7.2.2 and Section 7.2.3 have illustrated that making data structures variational leads to two different programming *modes* or *attitudes*. On the one hand, the focus can be on manipulating the underlying variant data structures that are represented, in which case the variational structure is maintained but not essentially changed. This is what Section 7.2.2 was all about. On the other hand, the focus can be on changing the variability within the variational data structure, in which case the existing represented objects are kept mostly intact. This is what Section 7.2.3 was concerned with.

Okasaki [1998] classifies the different ways of processing edits to data structures under the name of *persistence*. Imperative languages typically provide no innate support for persistence; that is, edits to data structures are destructive, making old versions inaccessible. In contrast, data structures in purely functional languages are by default fully persistent; that is, old versions are in principle always accessible as long as a reference to them is kept. (There are also the notions of partial persistence and confluent persistence that are not of interest here.)

Variational data structures add a new form of persistence that we call *controlled persistence* because it gives programmers precise control over what versions of a data structure to keep and how to refer to them. In contrast to the other forms of persistence (or non-persistence), which happen automatically, controlled persistence requires a conscious effort on part of the programmer to create and retrieve different variants of a data structure. Also uniquely, controlled persistence maintains the structure and relationship of all of the variants in the form of dimensions and choices, which the programmer can see and exploit.

7.3 Variational Software

A major motivation of the choice calculus was to represent variation in software. Having uncovered some basic principles of variational programming in Section 7.2, we are finally in a position to look at how we can put the choice calculus to work, through variational programming, on variational software.

As a running example we pick up the `twice` example that was introduced way back in Section 3.1. We will introduce a representation of (a simplified version of) Haskell as an object language in Section 7.3.1, together with a number of supporting functions. After that we will consider in Section 7.3.2 several simple example transformations for variational Haskell programs.

7.3.1 Representing Variational Haskell

Following the example given in Section 7.2.1 we first introduce a data type definition for representing Haskell programs, then extend it to make it variational. It is important to note that throughout this section we will be using Haskell as both the metalanguage of our DSEL and as an object language. The object language version of Haskell will not resemble Haskell syntax very closely, being represented instead as a (metalanguage) Haskell data type. Because of the technical limitations of the DSEL, described in Section 7.2.1, we have to make a number of simplifying assumptions and compromises in our definition of Haskell as an object language.

The major constraint is that within a data type definition the `V` type constructor can be applied to only one type. This has several implications. First, we cannot spread the definition of Haskell over several data types. We would have liked to have, for example, different data types for representing expressions and declarations, but since this is not possible, we are forced to represent function definitions using a `Fun` constructor as part of the expression data type. Second, ordinarily we would represent the parameter names of a function definition by simple strings. However, since we want to consider in our example the renaming of function parameters, and since we can't have subexpressions of both the type `V Haskell` and `V String`, we must push the representation of parameter names also into data type representing expressions. We will use smart constructors to ensure that we only build function declarations that use variable names as parameter names, but this representation is obviously less than ideal.

With these restrictions in mind, we give the relevant parts of our simple representation of Haskell in Figure 7.13. Assume in this definition that `Name` is just a type synonym for `String`. As before, we extend the representation of the object language with a new constructor, `VHaskell`, for embedding variational Haskell expressions into object language expressions. Thus, the

```
type VHaskell = V Haskell

data Haskell = App Haskell Haskell
             | Var Name
             | Val Int
             | Fun Name [Haskell] Haskell Haskell
               ...
             | VHaskell VHaskell
```

Figure 7.13: Definition of variational Haskell.

`Haskell` data type represents the expanded object language of Haskell, and the `VHaskell` type synonym represents variational Haskell.

Before we construct our `twice` example, we introduce a few more abbreviations and auxiliary functions to make working with variational Haskell programs more convenient.

First, we introduce a function that turns a string representation of a binary function into a constructor for building expressions using that function. For example, consider the simple Haskell expression `2*x`. When we try to represent this expression with the `Haskell` data type, we have quite some work to do. First, we have to turn `2` and `x` into Haskell expressions using the constructors `Val` and `Var`, respectively. Then we have to use the `App` constructor twice to form the application. In other words, we have to write the following expression.

```
App (App (Var "*") (Val 2)) (Var x).
```

The function `op` defined below performs all of the necessary wrapping for us automatically.

```
op :: Name -> Haskell -> Haskell -> Haskell
op f l r = App (App (Var ("(" ++ f ++ ")")) l) r
```

```
haskell :: Haskell -> VHaskell
haskell = Obj

choice :: Dim -> [Haskell] -> Haskell
choice d = VHaskell . Chc d . map haskell

(.+) = op "+"
(.*) = op "*"

x,y,z :: Haskell
[x,y,z] = map Var ["x","y","z"]
```

Figure 7.14: Auxiliary definitions for variational Haskell programs

Less importantly, this function also adds enclosing parentheses around the function name. This is exploited by the pretty printer to display operators with an infix representation.

In Figure 7.14 we define several other auxiliary functions and values that will make the definition of our `twice` example look nicer. First we introduce `haskell` as a synonym for `Obj`, as we did with `list` and `int`. We also provide a smart constructor `choice` for building choices within the `Haskell` data type more conveniently. We provide two infix operators `.+` and `.*`, defined in terms of `op`, that allow us to write infix addition and multiplication expressions of type `Haskell`. Finally, we provide three parameter names `x`, `y`, and `z`, for using within `Haskell` expressions.

Finally, we define a function `fun`, for defining top-level Haskell functions with an empty scope.

```
fun :: Name -> [Haskell] -> Haskell -> VHaskell
fun n vs e = haskell $ Fun n vs e emptyScope
  where emptyScope = Var ""
```

In the `Haskell` data type, each function declaration has a corresponding scope in which the function is defined. Since in our example we are only interested in the definition of `twice` and not its uses, we pass a dummy value `emptyScope` to complete the expression.

With all these preparations in place, we can now represent the variational Haskell function `twice` in our DSEL as follows.

```
twice = Dim "Par" ["x","y"]
      $ Dim "Impl" ["plus","times"]
      $ fun "twice" [v] i
      where v = choice "Par" [x,y]
            i = choice "Impl" [v .+ v, Val 2 .* v]
```

For comparison here is the original definition of the example given in Section 3.1.

dim $Par\langle x, y\rangle$ **in**
dim $Impl\langle plus, times\rangle$ **in**
`twice` $Par\langle$`x`,`y`$\rangle$ `=` $Impl\langle Par\langle$`x`,`y`$\rangle$+$Par\langle$`x`,`y`$\rangle$,`2`$*$$Par\langle$`x`,`y`$\rangle\rangle$

To check that this definition corresponds to the original, we can evaluate `twice` to view its pretty printed representation.

```
> twice
dim Par<x,y> in
dim Impl<plus,times> in
twice Par<x,y> = Impl<Par<x,y>+Par<x,y>,2*Par<x,y>>
```

To check that the definition actually represents the four variant implementations of the `twice` function that we expect, we can compute its semantics.

```
> psem twice
[Par.x,Impl.plus]  =>  twice x = x+x
[Par.x,Impl.times]  =>  twice x = 2*x
[Par.y,Impl.plus]  =>  twice y = y+y
[Par.y,Impl.times]  =>  twice y = 2*y
```

Looking at the definition of `twice` in our DSEL, notice that we have used Haskell's `where` clause to factor out parts of the definition. Whereas the definition of `i` is not really essential, the definition of `v` is needed to avoid repeating the choice in `Par`. In Chapter 4 we have seen how the **share** extension to the choice calculus enables common subexpressions to be factored out. In Section 7.1 we said that we do not include sharing in the `V a` data type. However, we can see here how Haskell's naming mechanisms can be reused to simulate these missing features (at least to some degree). Here is a slightly modified definition of `twice` that comes closer to the example using **share** given in Section 4.1.1.

```
twice = Dim "Par" ["x","y"] $
        Dim "Impl" ["plus","times"] $
        let v = choice "Par" [x,y] in
        fun "twice" [v] (choice "Impl" [v .+ v, Val 2 .* v])
```

But recall from Section 7.1 that there are some important differences between Haskell's `let` and the **share** and **macro** extensions to the choice calculus, the most significant of which is the potential for choice capture.

7.3.2 Edit Operations for Variational Haskell

As with variational lists, we also want to build variational Haskell programs incrementally. This is especially important for variational software since variability is often introduced to address changing requirements, introduce new features, or for other reasons that cannot be anticipated in advance.

In this section we will define several editing operations for variational Haskell programs that allow us to build up our `twice` example in several steps, starting from a single variant. While the example is simple, this process introduces several generic editing operations that could be implemented by a variation editor or development environment to support the evolution of variational software.

We begin with a single plain variant of the `twice` function with the parameter name `x`, implemented by the `+` operation.

```
xp = fun "twice" [x] (x .+ x)
```

Let us first consider the variation of the parameter name. In order to transform `xp` into a variational program where the parameter name can be either `x` or `y`, we need to do two things.

1. Add a declaration for the `Par` dimension.

2. Replace references to `x` by choices between `x` and `y`.

The first step is easy. It can be done by simply wrapping `xp` in a dimension declaration introduced with the `Dim` data constructor.

The second step requires a traversal of the abstract syntax tree representing the `twice` function, applying a transformation at each place where a variable `x` is encountered. This can be accomplished by employing the `everywhere` traversal function of the SYB library. All we need is to define a transformation that identifies occurrences of `x` variables and replaces them with choices. Such a transformation is easy to define.[4]

```
addPar :: Haskell -> Haskell
addPar (Var "x") = choice "Par" [x,y]
addPar e = e
```

[4]Here the fact that we have to represent parameters as expressions is to our advantage since we do not have to distinguish the different occurrences of variables (definition vs. use) and can deal with both cases in a single equation.

We can use this transformation as an argument for the `everywhere` traversal. Since `everywhere` is a generic function that must be able to traverse arbitrary data types and visit and inspect values of arbitrary types, the transformation passed to it as an argument must be a polymorphic function. The SYB library provides the function `mkT` that performs this task; that is, it generalizes the type of a function into a polymorphic one.

Combining both steps, we can define the following transformation `varyPar` on variational Haskell programs that introduces a new dimension `Par` and replaces every variable `x` in the original program with a choice in dimension `Par` between `x` and `y`.

```
varyPar :: VHaskell -> VHaskell
varyPar = Dim "Par" ["x","y"] . everywhere (mkT addPar)
```

We can confirm that `varyPar` has the desired effect by applying it to `xp` and observing the result.

```
> varyPar xp
dim Par<x,y> in
twice Par<x,y> = Par<x,y>+Par<x,y>
```

A limitation of the transformation as shown is that it renames *all* found variable names `x` and not just the parameters of `twice`. In this example, this works out well, but in general we have to limit the scope of the transformation to the scope of the variable declaration that is being varied. We can achieve this using the function `inRange` that we will introduce later.

The next step in transforming `xp` into the full-fledged `twice` example is to replace the addition-based implementation with a choice between addition and multiplication. This transformation works in essentially the same way, except that the function `addImpl` for matching and transforming the addition expression(s), has to do a more elaborate form of pattern matching.

```
addImpl :: Haskell -> Haskell
addImpl e@(App (App (Var "(+)") l) r)
          | l == r = choice "Impl" [e, Val 2 .* r]
addImpl e = e
```

With `addImpl` we can define a transformation similar to `varyPar` that declares a new dimension `Impl` and introduces choices through a combination of `everywhere` and `addImpl`.

```
varyImpl :: VHaskell -> VHaskell
varyImpl = Dim "Impl" ["plus","times"] . everywhere (mkT addImpl)
```

To verify the effect of `varyImpl` we can apply it directly to the plain variant `xp` and confirm that it produces a variational program that varies only in the `Impl` dimension.

```
> varyImpl xp
dim Impl<plus,times> in
twice x = Impl<x+x,2*x>
```

Or we can apply it to the variational program obtained from `varyPar xp` to produce the full `twice` example that varies in both the `Par` and `Impl` dimensions.

```
> varyImpl (varyPar xp)
dim Impl<plus,times> in
dim Par<x,y> in
twice Par<x,y> = Impl<Par<x,y>+Par<x,y>,2*Par<x,y>>
```

We can see that this result is not syntactically identical to `twice` since the dimensions are declared in a different order. However, if we reverse the order that we apply our two variation editing operations, we can verify that they indeed produce the same result as the hand-written definition for `twice`.

```
> varyPar (varyImpl xp) == twice
True
```

To take this example further, considering extending the parameter name dimension another option `z`, as we we did in Section 4.1.1. This transformation involves the following steps.

1. Extend the tags of the dimension declaration for `Par` by a new tag `z`.
2. Extend all `Par` choices that are bound by the dimension declaration by a new alternative `z`.

The first step is not difficult. It can be implementing using a similar strategy to the one we used throughout Section 7.2.3; that is, we can find and extract the dimension declaration, extend it with the new tag, then put it back using the context that we obtained during the extraction.

However, to change all bound choices is more complicated. This is because we do not know in advance how many choices are bound to `Par`, so we cannot easily use `extract` for this purpose. Additionally, we don't want to extend *all* choices in the `Par` dimension because we might end up extending choices that are not actually bound by the dimension that we extended.

To deal with both of these issues, we define a function `inRange` that applies a transformation to selective parts of a variational expression. Although the implementation of `inRange` is quite elegant and not very complicated, it is based on navigational function in the underlying SYZ library, so we don't show it here. Instead, we show only the type and argument names to support the discussion. The full implementation can of course be found in the accompanying source code.

```
inRange :: Data a =>
           (V a -> V a) -> (Pred a, Pred a) -> V a -> V a
inRange f (begin, end) e = ...
```

The inRange function takes three arguments: a transformation f, a pair of predicates on variational expressions, begin and end, and the expression e to transform. The predicates indicate the regions in e where f is to be applied. The expression e is traversed until a node nb satisfies the begin predicate. Then the transformation f is applied to every node descended from nb until a node ne is found that satisfies the end predicate. The transformation will continue applying f to other siblings of ne, but will not descend beneath ne. When all paths descending from nb have either been transformed or terminated by a matching end node, the traversal continues until another node satisfying begin is found. In this way, the predicates begin and end effectively act as switches to turn the transformation f on and off.

With the help of inRange we can now implement the transformation for extending a dimension. This function takes four parameters: the name of the dimension d, the new tag t, a function f to extend the bound choices, and the expression e in which to perform the update.

```
extend :: Data a => Dim -> Tag -> (V a -> V a) -> V a -> V a
extend d t f e = withFallback e $ do
    (c, Dim _ ts e') <- extract (dimDef d) e
    let e'' = f `inRange` (chcFor d, dimDef d) $ e'
    return (c <@ Dim d (ts++[t]) e'')
```

The extend function works as follows. First, it locates the declaration of dimension d and remembers the position in the context c. It then performs the extension of all choices bound by d by applying the function inRange to the scope of the found dimension, e'. Finding all relevant choices is accomplished by the two predicates passed as arguments to inRange. The first, chcFor d, finds choices in the scope of d, and the second dimDef d stops the transformation at places where another dimension declaration for d ends the scope. In this way the shadowing of dimension declarations is respected. Finally, we construct the result by inserting into c a dimension declaration with the new tag t and the transformed expression e''.

Now all we need to extend `twice` with a new possible parameter name `z` is a function for extending choices with new alternative expressions. This is accomplished with the following function `addAlt`.

```
addAlt :: V a -> V a -> V a
addAlt a (Chc d as) = Chc d (as ++ [a])
```

Putting it all together, we can extend our `twice` example with the new parameter name by employing `extend` and `addAlt` as shown below.

```
> extend "Par" "z" (addAlt (haskell z)) twice
dim Par<x,y,z> in
dim Impl<plus,times> in
twice Par<x,y,z> = Impl<Par<x,y,z>+Par<x,y,z>,2*Par<x,y,z>>
```

The pretty printed result shows the expected choice calculus expression.

The ability to programmatically edit variation representations is an important aspect of variational programming and our DSEL that we have barely scratched the surface of in this chapter. Identifying, characterizing, and implementing variation editing operations is an important area for future research since it directly supports the development of tools for creating, evolving, and maintaining variational software.

Chapter 8 – Related Work

Much of the work related to this thesis has been discussed already in Chapter 2. To briefly review, that chapter described the three main roles of languages in managing variation: *implementing* variability in the artifacts, *modeling* the variation space, and *configuring* individual variants. In the choice calculus, each of these roles is expressed through different constructs. Variation implementation is achieved by choices embedded in object structures, while local dimension declarations, and the nesting of dimensions within choices, support variation modeling. In the core choice calculus, the configuration of variants is supported externally by tag selection, but Chapter 5 introduced an extension for configuring choice calculus expressions from within the language itself.

Chapter 2 also introduced and qualitatively compared the three main approaches to implementing variation, called the *annotative*, *compositional*, and *metaprogramming* approaches. The complementary nature of the annotative and compositional approaches was explored in more depth in Section 6.1, focusing in particular on how they represent intended, structural feature interactions.

This chapter collects and discusses related research that has not been covered elsewhere in the thesis. Section 8.1 discusses the C Preprocessor, including how it is used and how it relates to the choice calculus and its extensions. Section 8.2 looks at the problem of organizing the variation space in more detail, comparing other approaches to dimensions in the choice calculus. Section 8.3 looks at other languages that provide internal support for configuration, comparing these to the **select** construct in Chapter 5, and discusses the problem of variant configuration more generally. Section 8.4 describes other attempts to combine the annotative and compositional approaches to variation implementation, comparing these to the compositional choice calculus in Chapter 6.

8.1 Comparison with the C Preprocessor

Although it is not state-of-the-art from a research perspective, the C Preprocessor [CPP] is by far the most widely used tool for managing static, code-level variability in real software. Therefore it is worthwhile to consider how CPP is used and how it relates to the choice calculus. Virtually every large-scale C or C++ software project uses CPP. In an empirical study of CPP usage in real C projects, Ernst et al. [2002] found that CPP directives make up an average of 8.4% of total lines of code (ranging from 4.5% to 22% in the 26 projects analyzed), and that 37% of C lines are enclosed by conditional compilation constructs.

Of course, CPP is used for many reasons besides representing variation. In particular, the management of header files relies on conditional compilation but is not related to variation in the sense we are interested in. A separate study of 40 projects by Liebig et al. [2010], focusing on CPP-implemented variation, found 23% of non-header source code to be variational on average, with variational code exceeding 50% in some projects. They also observed that the rate of variational code tends to increase as the overall size of the project increases, suggesting that variability increases in a software project over time. This study strongly supports the long-standing assumption that CPP is widely and heavily used for managing variation in practice.

In many ways, CPP exemplifies the benefits of annotative variation languages, which probably explains its success. Any text-based artifact can be arbitrarily varied, making it flexible and expressive. The ability to work with multiple different object languages makes it possible to, for example, synchronize variation in source code and documentation.

However, CPP is also a very unstructured language that is widely considered a common source of errors [Spencer and Collyer, 1992, Favre, 1995, Ernst et al., 2002]. The ability to arbitrarily vary text can lead to situations where only some variants are even syntactically correct, let alone type or semantically correct. With standard build tools, a bad variant can only be detected by individually generating, compiling, and testing it. Since the number of variants grows exponentially with respect to the number of CPP macros used in conditional compilation directives, it is almost always impossible to generate and test each variant in this way.

A formal notion of "disciplined" CPP use was introduced by Kästner et al. [2009b] to identify a subset of CPP usage that is guaranteed to produce syntactically valid variants. A study by Liebig et al. [2011] found that approximately 86% of existing uses of CPP are disciplined. (Note that not all undisciplined usage leads to syntactically incorrect variants since the definition is conservative.) By operating on the level of abstract rather than concrete syntax (see Section 3.3.1), the choice calculus avoids the problem of syntactically incorrect variants altogether—a syntactically valid choice calculus expression can only produce syntactically valid variants in the object language. This is a quality shared also by the CIDE tool [Kästner et al., 2008a], which restricts variation to syntactically optional elements in the object language syntax. The differences between the choice calculus's alternative-based model of variation and CIDE's optionality-based model have been discussed in Section 3.2.3.

A loose correspondence between the concepts and constructs of the choice calculus and those in CPP are summarized in Figure 8.1. The first row in the table contrasts the view of the object language from within each metalanguage, as described above. Recall from Section 3.3.1 that the choice calculus can also be instantiated by a particular object language, replacing the generic AST representation of object structures with the specific abstract syntax of the object language.

The next row shows a correspondence between choice calculus tags and CPP macros (a *macro* in this sense is just a variable used by CPP conditional

Choice Calculus	C Preprocessor
$a\prec e_1,\ldots,e_n\succ$	Plain text
Tags: t, u	Macros: `T`, `U`
dim $D\langle t,u\rangle$	N/A
$D\langle e_1,e_2\rangle$	`#if T` / e_1 / `#elif U` / e_2 / `#endif`
macro v := **dim** $D\langle t,u\rangle$ ...	N/A
share v := **dim** $D\langle t,u\rangle$ ...	`#if T` / `#define V ...` / `#elif U` / `#define V ...` / `#endif`
select $D.t$	`#define T 1`

Figure 8.1: Correspondence of choice calculus and CPP concepts.

compilation directives). In the choice calculus, a configuration is identified by a particular selection of tags; in CPP a configuration corresponds to a particular definition of the macros used in conditional compilation directives.

Dimensions impose a structure on the configuration space that is not present in CPP, and choices ensure that this structure is respected at each variation point. Note that the concept of a dimension is independent of and more general than the dimension declaration construct **dim**. This is discussed in more depth in Section 8.2. CPP has no corresponding means of organizing macros into related groups. However, external representations can be used along with CPP to organize the variation space, such as feature diagrams [Kang et al., 1990] or the Linux Kconfig tool [She et al., 2010]. These have the advantage of being more expressive than dimensions (since they can usually model arbitrary relationships between tags). However, unlike choices and dimensions in the choice calculus, the consistent usage of CPP macros in conditional compilation constructs is not enforced, which is known to lead to bugs in practice [Tartler et al., 2009]. Relatedly, recall that CPP's conditional compilation directives are more flexible but less structured than choices since they are based on arbitrary boolean inclusion conditions—the trade-offs involved here have been discussed in more depth in Section 3.2.3.

CPP provides no mechanism for the explicit reuse of variational components. That is, there is no feature that corresponds to the **macro** extension introduced in Chapter 4. At this point we need to distinguish between the two different roles of macros in CPP. One role is as variables in conditional compilation constructs, which we have considered so far. We can call these more specifically *configuration macros*. The other role is closer to the traditional use of the word "macro", to represent text that is expanded in place of the macro name; we will call these *text macros*. In CPP, the choice calculus **macro** construct might correspond to expanding a text macro in multiple places, with different settings of its enclosed configuration macros each time. This is not possible since any conditional compilation directives in a macro definition are resolved at the point of definition. This limitation of CPP is addressed by the Boost Preprocessor [Karvonen and Mensonides, 2001, Abrahams and Gurtovoy, 2004, p. 281], which provides many other features and improvements over standard CPP for definining text macros. CPP can, however, simulate the **share** extension in a relatively straightforward way by wrapping a text macro definition in conditional compilation directives.

Both the choice calculus and CPP support external configuration. In the choice calculus this is realized by tag selection, while in CPP macros can be set at the command line or in a Makefile. Like the **select** extension in Chapter 5, the CPP `#define` directive also provides syntactic support for configuration within the language itself. However, `#define` is much more complicated than **select** since (1) macros can be conditionally defined, (2) macros can be defined in terms of other macros, (3) the value of a macro can change during preprocessing, and (4) the order that files are processed is significant. This complexity makes it difficult to reason about variability in CPP-annotated code [Kästner et al., 2011a]. Consider the example in Figure 8.2, which shows the interaction of two features defined by the macros

```c
void launch() {
#if SECURE
  authorize();
#endif
  initiate();
}
```

a. nukes.c

```c
#if VERY_SECURE
#define SECURE 1
#endif
```

b. security.c

```c
File* data() {
#if SECURE
  authorize();
#endif
  return d;
}
```

c. secrets.c

Figure 8.2: Ambiguity of internal configuration in CPP.

`SECURE` and `VERY_SECURE`. The file `security.c` declares that any program that includes the `VERY_SECURE` feature also includes the `SECURE` feature by defining `SECURE` to `1`. However, since macros are globally scoped and their values can change during preprocessing, this configuration depends crucially on the order that the files are processed. If `SECURE` is initially unselected and the files are processed in the order shown in the figure, then the security functionality will be included in `secrets.c` but *not* in `nukes.c`, even though they both refer to the same macro. The **select** extension in the choice calculus provides a simpler, more structured view of configuration since a tag can be selected precisely once and this selection will affect all choices bound to the corresponding dimension.

8.2 Organizing the Variation Space

While variation implementation has been discussed in depth in Chapter 2 and Chapter 6, the more abstract problem of *organizing* or *modeling* the variation space has received comparatively little attention in this thesis. This section will discuss work in this area, focusing in particular on the large body of work on *feature modeling* [Lee et al., 2002] as used in the software product line (SPL) [Pohl et al., 2005] and feature-oriented software development (FOSD)

[Apel and Kästner, 2009] communities. Although feature modeling is only partly about describing variability, that is the role we will focus on here.

Feature models describe variation in a system at a very high level in terms of abstract features which may be included or not in a program. In contrast, the choice calculus is more focused on *how* that variation is realized in the artifact. However, the choice calculus does provide basic mechanisms for organizing the variation space in terms of *dimensions*.

It is important to separate the concept of a dimension from the dimension declaration construct **dim**. In instances of the choice calculus that do not include the **dim** construct (such as variational types [Chen et al., 2012, 2013]), each choice still refers to a dimension by name, and these dimensions still structure and synchronize choices. Without **dim**, the namespace of these dimensions is just uncontrolled, and the tags are unnamed.

While dimensions provide essential structure to the choice calculus, they are somewhat limited compared to some of the feature modeling techniques described in this section. Therefore we can imagine complementing a choice calculus-based implementation with an external representation of the variation model. This would impose additional constraints on dimensions and tags not encoded in the choice calculus representation itself.

Feature models are represented in a variety of ways. At the simplest end of the spectrum is the applicative language of the GenVoca model of SPL construction [Batory and O'Malley, 1992]. GenVoca's modeling language consists of two kinds of elements: a program is represented as a constant, and a feature is represented as a function that takes a program and produces a new program with the feature included. The only operation is function application. Valid combinations of features are simply enumerated, though naming can be used to abstract out common parts, leading to a hierarchical description of the variants that can be produced.

The AHEAD tool suite Batory et al. [2004] generalizes the GenVoca model to include sets of elements called *refinements*. It generalizes feature application to the refinement composition operator described in Section 6.1.1, which is overloaded to implement different refinement operations for different kinds of artifacts. This operator is reused in the semantics of the compositional choice calculus in Section 6.5. The culmination of this approach is the feature algebra described by Apel et al. [2008b], which describes feature composition in terms of tree superimposition and describes its algebraic properties.

Since products are built up incrementally by applying features to programs, the feature algebras described above provide direct support for compositional feature implementations with ordering constraints. Their main drawbacks are that each variant must be enumerated and ordering constraints are not explicitly represented or enforced. Section 6.2 shows how the choice calculus can be used to describe a variational feature algebra term so that individual variants can be generated by first selecting a variant term, then composing the variant accordingly. This also means that the ordering constraints must be considered only once in the variational expression and will then be preserved in each selected variant.

Feature diagrams are an alternative, graphical notation for modeling the relationships between features [Kang et al., 1990]. Although many different notations and extensions have been developed [Schobbens et al., 2006], the core language is relatively simple. In a feature diagram, features are arranged hierarchically such that children are dependent on their parents. A feature can be marked as mandatory or optional, and groups of features with a common parent can be joined together as inclusive or exclusive alternatives. This language is equivalent to several other representations of feature models, including propositional formulas and tree grammars [Batory, 2005].

The basic relationships in feature diagrams loosely correspond to patterns in the choice calculus. Dependencies between mandatory features correspond to nested object structures, while dependencies between optional features correspond to nested choices. Exclusive alternation corresponds to a dimension, while optionality can be represented by a dimension with two alternatives, one that includes the feature and one that does not. Inclusive alternation can also be expressed by nested choices, but this requires either the duplication of alternatives or use of the **share** extension from Chapter 4.

Other feature diagram relationships are less straightforward to express in the choice calculus. For example, most feature diagram notations allow inclusion and exclusion conditions between arbitrary features called *cross-tree constraints*. Simple cross-tree constraints can be simulated in the choice calculus with the **select** construct from Chapter 5—by nesting a **select** expression within a choice of another dimension, we can enforce constraints between otherwise unrelated dimensions. However, some feature diagrams also allow constraints involving numeric values which have no direct correspondence to the choice calculus.

Unlike the applicative feature algebras described above, feature diagrams are usually not a processable part of a SPL system but rather employed as design documents and developer specifications. In other words, the constraints they describe are not enforced, leaving room for error in their implementation. Although choice calculus dimensions are somewhat less expressive than other variation modeling techniques, they have the advantage of directly constraining the implementation. There has been quite a bit of work on ensuring that variation implementations are consistent with the feature models that describe them [Czarnecki and Pietroszek, 2006, Metzger et al., 2007, Thaker et al., 2007]. A common class of problems is a dependency between features at the implementation level that is not reflected at the model level. Usually this dependency is only discovered when a variant that does not satisfy the dependency is generated. Kästner et al. [2009c] describe how to detect and resolve such cases.

The Linux Kconfig tool [Kconfig, She and Berger, 2010] is used to organize the configuration space of the Linux kernel, making it a prominent example of variation modeling in a large-scale, real-world application [She et al., 2010]. Kconfig uses a textual language that describes the type of each configuration option and the dependencies between them. Like feature diagrams, Kconfig's variation model is not enforced in the implementation of the kernel, which leads to inconsistencies in practice [Tartler et al., 2009].

8.3 Internalized Selection

Besides variation modeling and implementation, the third role of variation languages is to support the configuration, generation, or selection of individual variants (see Section 2.2). In the choice calculus, configuration is usually considered to be an external operation. However, in Chapter 5 we introduced an extension that internalized selection, making it an implementation-level feature of the language itself. In this section, we look at work related to this extension.

Some of the variation modeling languages described in the previous section double as configuration languages. For example, AHEAD's feature algebra describes how individual variants should be assembled from components. Similarly, the Kconfig tool is used to drive the process of configuring a particular variant of the Linux kernel. However, these tools are purely external to the languages used to implement the variability at the code level, such as refinements and CPP.

More research has been done on external configuration at the modeling level. For example, Czarnecki et al. [2005] enumerate the ways that variation can be reduced in cardinality-based feature models. This effectively yields a catalog of potential configuration operations, of which our **select** construct is just a single instance. Subsequently, Classen et al. [2009] have provided a formal semantics for the staged configuration of feature models.

Internalized selection has also been explored in some depth at the modeling level. Most relevant is research on *nested* SPLs (also called *dependent* or *multi*-SPLs) [Krueger, 2006, Rosenmüller et al., 2008, Reiser, 2009, Haber et al., 2011]. This research extends variation modeling languages with constructs to represent the reuse of entire SPLs, their (potentially partial) configuration, and configuration constraints between them. For example, Reiser [2009] shows how a UML-based feature modeling language can be extended with a notion of a *composite component* defined by another feature model, and *configuration links* that define configuration decisions or constraints between components. In this way a multi-SPL is described by a hierarchical feature model. Rosenmüller and Siegmund [2010] describe how to automatically configure multi-SPLs based on such hierarchical models, bridging the gap between modeling and configuration.

In the choice calculus, the reuse of variational components is supported at the implementation level by the **macro** extension, while the ability to internally configure these components was one of the major motivations for the **select** extension (see Section 5.2). Recall from Section 8.1 that CPP also supports internal configuration at the implementation level through the `#define` directive, while the Boost Preprocessor supports the reuse of variational components. There we argue that the **select** extension provides a more structured and disciplined solution to the problem.

Another recent example of implementation-level internal configuration can be found in the "variability" DSEL[1] implemented in the SugarJ language [Erdweg et al., 2011a,b]. The DSEL adds syntactic support to Java for describing variation points, defining feature models, reusing variational components, and locally configuring these components by individually enabling or disabling features.

8.4 Combining Annotative and Compositional Variation

The trade-offs between the annotative and compositional approaches to feature implementation were discussed in depth in Section 6.1, which focused especially on how they represent intended, structural feature interactions.

[1]No corresponding publication or release, but available online at: `https://github.com/seba--/sugarj/tree/master/case-studies/fosd`. Discovered through private communication with Sebastian Erdweg.

Many researchers have attempted to take advantage of the complementary nature of these trade-offs by combining the two approaches into a single representation, as we did with the compositional choice calculus (ccc) in Chapter 6. In this section, we focus on some of these other attempts, comparing them to ccc and to the choice calculus in general.

Kästner and Apel [2008] have suggested that their annotative CIDE tool [Kästner et al., 2008a] could be integrated with the compositional AHEAD tool suite [Batory et al., 2004] to achieve the benefits of each. They also discuss the implications of such a merger, many of which we have not considered here, for example, that an integrated model can support migrating from one implementation approach to another (which might be especially desirable if migrating away from CPP, for example). Elsewhere, they propose the idea of a "virtual separation of concerns", which attempts to bring the maintenance and understandability benefits of separability in compositional approaches to annotative approaches through tool support for working with projections of annotated artifacts [Kästner and Apel, 2009].

Kästner et al. [2009a] have also created LJ^{AR}, a formal language for combining annotative and compositional variation in Lightweight Java programs (a formal subset of Java [Strniša et al., 2007]). While the semantics-preserving transformation laws of the choice calculus and ccc describe the commutation of annotative variation (in the form of choices) with and within compositional components (object structures), LJ^{AR} supports refactorings for moving *between* the two implementation approaches—that is, moving from an annotative representation to a compositional one, or vice versa. Another difference is of course that LJ^{AR} is specific to the object language of Lightweight Java.

The *XML Variant Configuration Language* (XVCL) [Zhang and Jarzabek, 2004] is another language-based attempt to merge the annotative and compositional approaches. Like CPP (but unlike the choice calculus and CIDE), its in-place variation annotations are structurally undisciplined (that is, they vary plain text rather than abstract syntax). Distributed variation is supported through named “breakpoint” annotations, where code specified elsewhere can be automatically inserted. While these breakpoint-controlled insertions provide separability, the need to insert breakpoint annotations means that XVCL does not support stepwise refinement, a core tenet of compositional approaches. (However, the sometimes necessary “hook” method technique [Liu et al., 2006] violates this in purely compositional approaches as well.) This makes separability in XVCL more similar to **share** and **macro** expressions in the choice calculus than to compositional approaches.

Chapter 9 – Conclusion

This thesis has presented the choice calculus as a fundamental representation of variation in software and other artifacts. The choice calculus is intended to fulfill a role in variation research similar to the lambda calculus in programming language research. If successful, it can improve the state of variation research in at least three ways. First, it can provide a *common language of discourse* for researchers applying variation to different problems and in different contexts, supporting the precise and effective communication of ideas between fields. Second, it can provide a *shared research platform* that supports the reuse of theoretical machinery and results, lowering the barrier of entry for the description of new advanced variation features and sophisticated analyses. Third, it can support the *development of tools* by providing an established core of representations and operations to build on.

Section 9.1 discusses evidence that the choice calculus can achieve these goals by describing some successful applications of the language and touching on how the choice calculus is already influencing the way other variation researchers think and talk about variation. Section 9.2 briefly reviews the most important contributions of this thesis, and Section 9.3 provides some immediate directions for future work.

9.1 The Choice Calculus in Practice

The choice calculus is designed with a long-term vision of supporting variation research by providing a language that is simple and general enough to support formal theoretical research, but that is also customizable enough—by extending it with different language features and instantiating it with different object languages—to support a broad range of sophisticated applications.

In Section 9.1.1 we describe how the choice calculus helped us to solve the problem of extending type inference to variational programs. In Section 9.1.2 we describe other applications of the choice calculus and share some quotes that illustrate the impact that the choice calculus has had already.

9.1.1 Application to Variational Typing

Our work on variational typing [Chen et al., 2012, 2013] represents the most substantial application of the choice calculus so far. More importantly, it demonstrates the advantages of the choice calculus as a research platform by directly reusing many of the ideas, formalisms, and results from this thesis.

The problem of typing variational software is important and difficult. The traditional view of software product lines assumes that *application engineers* will configure, supplement, compile, and test an individual variant before distributing it to a client [Pohl et al., 2005, p. 30]. This makes the problem of ensuring the type correctness of each variant manageable (although still potentially costly) since type errors can be resolved before the variant is ever released. However, a lot of variational software, especially in an open source setting, is distributed directly to clients who will configure and compile their own variant including precisely the features they need. A type error (or other fault) that appears at this point is too late since the client is not in a position to resolve or even understand the error.

Therefore, we want to ensure that *every variant* that can be generated from a variational software project is well typed. The brute-force strategy of generating all variants and typing each one individually is intractable since the number of variants grows multiplicatively with respect to the arity of each independent dimension of variation. The solution is to type the variational program directly such that if the variational program is well typed, every variant that can be selected from it is also well typed. Other researchers have also addressed the problem of type checking variational software efficiently [Thaker et al., 2007, Kenner et al., 2010, Kästner et al., 2012a]. Our work

$$\begin{array}{rcll} T & ::= & \tau & \textit{Constant Types} \\ & | & a & \textit{Type Variables} \\ & | & T \rightarrow T & \textit{Function Types} \\ & | & D\langle T, T\rangle & \textit{Choice Types} \end{array}$$

Figure 9.1: Variational types.

distinguishes itself by solving the more difficult problem of *type inference*, which leads to additional subtle issues, such as how to *represent* the type of a variational program.

As the basis for this work, we use the *variational lambda calculus* (VLC), which is just the choice calculus instantiated by the object language of the lambda calculus, as described in Section 3.3.1. Already, we can reuse many definitions from Chapter 3, such as the (instantiated) equivalence laws and the formal semantics. For example, we can concisely express the important property that our type system should support in terms of the semantics of the choice calculus: if e is well typed, then $\forall e' \in rng(\llbracket e \rrbracket)$, e' is well typed.

The next question is how to represent the types of VLC expressions. Since different plain variants can have different plain types, the type of a VLC expression must also be variational. Therefore, we use a second instantiation of the choice calculus with the object language of types to produce the language of *variational types* shown in Figure 9.1.

For each well-typed VLC expression e, our type system assigns a *correspondingly* variational type T. The judgment $\Gamma \vdash e : T$ states that e has type T in environment Γ. Since the languages of e and T are both instances of the choice calculus, we can reuse their respective semantics to precisely capture the relationship between e and T in the following theorem.

Theorem. *If* $\Gamma \vdash e : T$ *and* $(\delta, e') \in \llbracket e \rrbracket$, *then* $\Gamma \vdash e' : T'$ *where* $(\delta, T') \in \llbracket T \rrbracket$.

This theorem describes the most important property of our type system, which is that typing relations are preserved through the process of selection.

$$\textsc{T-App}\quad \frac{\Gamma \vdash e : T'' \qquad \Gamma \vdash e' : T' \qquad T'' \equiv T' \to T}{\Gamma \vdash e\, e' : T}$$

Figure 9.2: Typing rule for applications in VLC.

More plainly, it states that for any variant e' of e, we can obtain its type T' from T by simply applying the same decision to both e and T. Since the variability in the two languages is expressed by the same metalanguage of the choice calculus, we can describe this property quite simply in the theorem in terms of the definitions in Chapter 3. The proof of this theorem (and many other results) demonstrates the suitability of the choice calculus for rigorous formal work [Chen et al., 2013].

The type system also makes critical reuse of the equivalence laws from Section 3.5, instantiated for variational types. When typing an application $e\, e'$ in the (plain) lambda calculus, we usually require that (1) the type of e is a function type, $T' \to T$, and that (2) the type of e' matches its argument type, T'; then the result of the whole application has type T. However, these requirements on e and e' are too strict for VLC since either or both of the expressions can be variational. What we want to say instead is that the application can "never go wrong" in any of the variants. That is, if we apply the same decision to both e and e', we will get plain lambda calculus expressions that satisfy the requirements above. We can express this property precisely using the equivalence relation from Section 3.5, by requiring that if e has type T'' and e' has type T', then it must be the case that $T'' \equiv T' \to T$. This requirement is shown in the context of the actual typing rule in Figure 9.2.

The type inference algorithm for VLC is an extension of algorithm $\mathcal{W}$ by Damas and Milner [1982]. The extension consists mostly of an equational unification algorithm for variational types that supports the more flexible typing of function applications shown in Figure 9.2. The equational theory is defined by the equivalence laws in Section 3.5, instantiated for the object language of variational types. The inference algorithm is more efficient than a brute-force implementation since choices capture variation locally within types (so we must only type their surrounding context once), and since the Chc-Idemp equivalence law allows us to eliminate choices when two alternatives have the same type [Chen et al., 2013].

In subsequent work we have extended this approach to be error-tolerant in the sense that we can infer *partial* variational types that may contain type errors in some variants [Chen et al., 2012]. This is useful not only for locating type errors, but also for supporting the incremental development of variational software. The implementation of this extension relies on computing the *error pattern* associated with an expression. Error patterns are represented by yet another instantiation of the choice calculus, this time with the simple object language of boolean values, which represent whether a variant contains a type error or not.

Our work on variational typing demonstrates that the choice calculus is a suitable metalanguage for rigorous theoretical research involving variation. Perhaps more importantly, it illustrates the value of foundational research in this area, and its potential for reuse. When we first designed the choice calculus and identified (and proved correct) the set of semantics-preserving equivalence laws [Erwig and Walkingshaw, 2011b], we did not have any particular application in mind. However, the equivalence laws turned out to be the key to the unification of variational types, and so we were able to reuse them. If not for this earlier work in a more general setting, we might have arrived instead at a more specific solution to variational type unification, which could not be easily reused in other contexts.

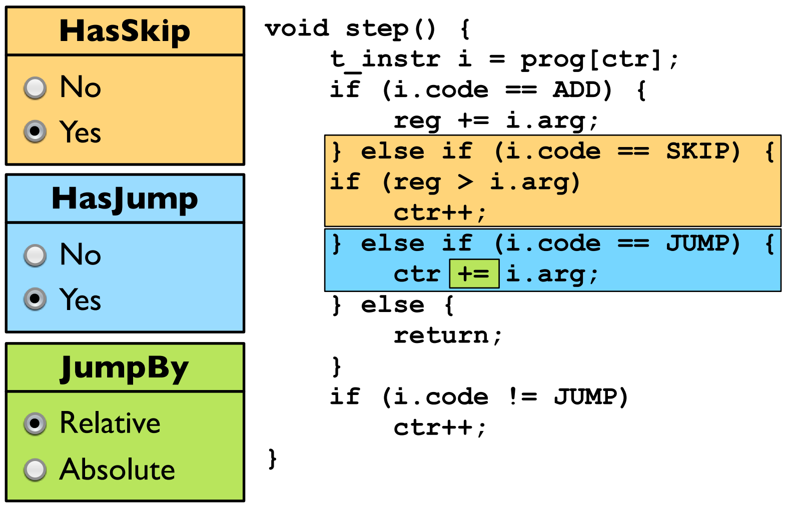
9.1.2 Other Applications

In Chapter 7 we applied the choice calculus to the development of a domain-specific language for exploring the idea of *variational programming* [Erwig and Walkingshaw, 2012a]. This work led to design questions and insights that support the development of variational data structures (see Section 9.3). For example, it posed the question of how best to embed variation in an algebraic data type, and presented one possible solution in Section 7.2.1.

The work on variational programming advocated a view of *computing* with variation and mixing variation into all kinds of representations. This contrasts with the typical view of variation as something that is resolved statically, for example, configuring a software product line into a single product which is only later executed. Kästner et al. have stated that their solution to the problem of testing variational software more efficiently was "inspired by variational programming by Erwig and Walkingshaw" [2012c]. Their approach is to use a *variational interpreter* that efficiently tracks the evaluation of all variants simultaneously, allowing them to execute regular unit tests (that is, unit tests that do not say anything about variation) on a variational program directly.

We have also applied the choice calculus in the context of a user study on *understanding* variation in software [Le et al., 2011]. Specifically, we developed an interactive, visual representation of choice calculus expressions, then tested two kinds of understanding against a corresponding representation in CPP. First, we tested users' ability to understand the behavior of a single program variant. Second, we tested their ability to understand the overall space of potential configurations, determined by counting the number of valid and unique program variants.

The visual representation, shown in Figure 9.3, shows just one variant at a time. This is intended to support a "virtual separation of concerns" (VSoC) [Kästner and Apel, 2009] by allowing programmers to think, as much as possible, in terms of the plain program that they are currently interested in.



HasSkip: No [✓ Yes]

HasJump: No [✓ Yes]

JumpBy: [✓ Relative] Absolute

```
void step() {
    t_instr i = prog[ctr];
    if (i.code == ADD) {
        reg += i.arg;
    } else if (i.code == SKIP) {
    if (reg > i.arg)
        ctr++;
    } else if (i.code == JUMP) {
        ctr += i.arg;
    } else {
        return;
    }
    if (i.code != JUMP)
        ctr++;
}
```

Figure 9.3: Graphical representation of a choice calculus expression.

Dimensions are represented by colored panels in the side bar, with radio buttons corresponding to each tag. Choices are represented by colored text in the main text area, where the color corresponds to its dimension and the text is the alternative corresponding to the currently selected tag.

The results of the study were that users were able to more quickly and more accurately answer questions targeted at both kinds of understanding described above, by statistically significant amounts. This supports claims not only about our prototype visualization, but also about related representations that also rely on background colors to represent variability, such as FeatureCommander [Feigenspan et al., 2011], and that support VSoC, such as CIDE [Kästner et al., 2008a].

More importantly from the perspective of this thesis, it presents another quite different application of the choice calculus, illustrating how it can support the development of tools and other kinds of research on variation.

9.2 Summary of Contributions

The main contribution of this work is the *choice calculus*, a formal language for describing variation. In Chapter 3 we presented the choice calculus, its denotational semantics, a set of semantics-preserving equivalence laws, and a variation design theory supporting the elimination of redundancy in choice calculus extensions.

The core calculus is *generic* in the sense that it is based on a simple tree model of the artifact being varied. It is also intentionally *minimalistic*, providing only two constructs in its simplest form: one for encoding the tree structure of the object language, and one for capturing points of variation within this tree (choices). These qualities support analytically rigorous work by eliminating special cases introduced by a particular object language, and by minimizing the number of constructs that must be considered when introducing a new feature or property.

In order to support the broadest range of applications, however, the core calculus can also be *instantiated* by specific object languages (Section 3.3.1). This allows researchers to explore variation in the context of a particular artifact type or research question. For example, in Section 9.1.1 we showed how instantiating the choice calculus twice, once by lambda calculus and once by its types, supported research on the interaction of variation and type inference. Importantly, the instantiation process can be extended to other definitions, such as the equivalence laws, enabling their reuse with different object languages with relatively little additional effort.

In order to support research at the level of variation metalanguages, the choice calculus can also be *extended* with new language features. The semantics of the choice calculus is designed so that many language extensions can be defined modularly, requiring minimal changes to existing definitions. We have presented examples of three such extensions here (in Chapters 4 and 5), which can serve as templates for other researchers to add new features to the choice calculus in order to support their own work.

In Section 5.5 we presented a *configuration type system*. The type system ensures that a choice calculus expression is well formed and associates with that expression a configuration type that describes all of the decisions one can make about the expression in order to resolve it into a plain variant.

In Chapter 6 we presented the *compositional choice calculus* (CCC). A language based derived from the choice calculus that integrates that annotative and compositional approaches to variation implementation.

Finally, in Chapter 7 we introduce the idea of *variational programming* and a DSEL to support its exploration. The idea is that variation is not just a static quality of software product lines, but something that can be computed with. In principle, we can imagine making any data type variational and "lifting" all of operations based on that data type so that they preserve the variability of their arguments.

9.3 Future Work

Since the long-term vision of this thesis is to support all kinds of variation research, we can imagine many applications for the choice calculus. A measure of the success of this work will be if most of these ideas are pursued by researchers that are not us. However, in this section, we will discuss a few immediate *applications* of the choice calculus, and *extensions* to the choice calculus that we plan on pursuing.

The idea of variational data structures, introduced in Chapter 7, has a huge range of potential applications. The most immediate applications are to support analyses on software product lines, such as variational graphs to support control flow analyses [Bodden et al., 2013], and variational trees to support parsing and type checking [Kästner et al., 2011b, 2012a]. However, there are many other other applications for variational data structures not concerned with software product lines. For example, navigation software must compute several possible routes through a transportation network that can change depending on road closures and user-defined settings—this can be viewed as a query over a variational graph that produces a variational path as a result.

In our work on variational typing, we found that the local dimension declarations (introduced by the **dim** construct) cannot be maintained in variational types and pose a challenge for typing even when only part of the term language of VLC. A detailed description of the problem is outside the scope of this discussion but can be found, along with a partial solution, in our previous work [Chen et al., 2013]. Since the alternative of global scoping is undesirable for other reasons, this has led us to explore alternative means of introducing and scoping new dimension names. One possibility is *module scoping*. That is, we can extend the choice calculus with a new construct for introducing a module boundary, and associating with that module several dimension names and possibly the relationships between those dimensions. Within the module, we do not have to deal with the complexity of local dimension declarations, but the namespace is still managed explicitly.

A module system can provide other advantages too. For example, it can provide an abstraction boundary limiting the scope of selections, macros, and other features that have potentially far-reaching effects. We might also be able to express more sophisticated relationships between dimensions than is possible by nesting dimension declarations in choices. We have explored a very simple module system already, in the context of the selection operation [Erwig et al., 2013a], but it does not provide all of the features described here.

Finally, in Section 3.2.3 we discussed the design decision to base the choice calculus on an alternative-based model of variation. We are currently working on generalizing the notion of a choice in order to accommodate other models, such as the one employed by Kästner et al. [2011b] that associates arbitrary boolean expressions of tags with each subexpression of a choice. On the one hand, this loses some of the simplicity of the choice calculus, but on the other hand, it is more flexible and useful in some applications (such as typing CPP-annotated code). Generalizing the representation of variation points will allow us to more easily interact with and support other variation representations that already exist, increasing the applicability of the choice calculus.

Appendix A – Proof of Equivalence Laws

This appendix presents a mechanized proof of Theorem 3.5.1 for the unextended core choice calculus, instantiated with binary trees, and restricted to binary choices. The proof is written in the language of the Coq proof assistant [Bertot and Castéran, 2004]. The source code of the proof is available online.[1]

The variant of the choice calculus used in this appendix is very simple. This makes the task of writing automatically verifiable proofs easier, but also means that we can only prove a subset of the equivalence rules presented throughout the thesis. This is a typical sort of trade-off in tool-supported theorem proving. However, the rules that we can prove with this simplified variant—related to the commutation, introduction, and elimination of choices—are the most interesting from the perspective of developing a general theory of variation. They are also the most generally applicable since they can be used with every variant of the choice calculus.

Finally, this appendix does not describe Coq syntax or explain the proofs in-depth. Coq proofs are developed interactively and have very limited explanatory value. However, Section A.1 and Section A.2 describes the representation of the choice calculus syntax and semantics, respectively, used in the proof, and Section A.3 relates each proof to its corresponding equivalence rule in Section 3.5.

A.1 Syntax Definition

Dimension names are mapped to the domain of natural numbers, and tags (since we only consider binary dimensions) are mapped to the domain of

[1] `https://github.com/walkie/CC-Coq`

boolean values. The synonyms `L` and `R` refer to the left tag and the right tag of a dimension, respectively.

```
Definition dim := nat.
Definition tag := bool.
Definition L : tag := true.
Definition R : tag := false.
```

The syntax of the choice calculus is defined as a simple inductive data type with three constructors. The `chc` constructor represents a binary choice in the given dimension, while `leaf` and `node` represent components in the object language of binary trees of natural numbers.

```
Inductive cc : Type :=
  | leaf : nat -> cc
  | node : nat -> cc -> cc -> cc
  | chc  : dim -> cc -> cc -> cc.
```

We also declare a separate inductive data type for representing plain trees in our object language. This clear separation, at the type level, of variational and plain binary trees will make reasoning about the denotational semantics of a variational expression much easier.

```
Inductive tree : Type :=
  | t_leaf : nat -> tree
  | t_node : nat -> tree -> tree -> tree.
```

A.2 Semantics Definition

The denotational semantics of a choice calculus expression is a function from selections (decisions) to plain binary trees.

```
Definition denotation := selection -> tree.
```

A selection is a total function from dimension names to the selected tags. This is a slightly different view of decisions than presented in Section 3.4, where we used lists of qualified tags. This representation ensures that any selection will fully resolve a variational expression to a plain variant, once again simplifying the proofs.

```
Definition selection := dim -> tag.
```

The source code linked to above provides several functions for constructing selections. These are omitted here since they are not needed for the proofs.

Finally, the semantics function maps variational expressions to semantic denotations. This is just a straightforward recursive function.

```
Fixpoint sem (e:cc) : denotation := fun s =>
  match e with
    | leaf a     => t_leaf a
    | node a l r => t_node a (sem l s) (sem r s)
    | chc  d l r => if s d then sem l s else sem r s
  end.
```

A.3 Proofs that the Laws Preserve the Semantics

Two expressions are semantically equivalent if they yield the same plain trees for all possible selections. The definition of `equiv` defines this property, while the subsequent notation definition introduces an operator <=> that we can can use to represent semantic equivalence. This corresponds to the $\equiv$ operator used throughout this thesis.

```
Definition equiv e1 e2 := forall s, sem e1 s = sem e2 s.
Notation "e1 <=> e2" := (equiv e1 e2) (at level 75).
```

With this property, we can now state and prove many of the semantic equivalence laws presented in Section 3.5.

Note that since we are using a simpler syntax, only a subset of the rules are applicable, namely Chc-Idemp, Chc-Chc-Merge, Chc-Chc-Swap, and Chc-Obj. We also prove that the semantic equivalence relation is reflexive, symmetric, and transitive. Finally, the congruence rule from Section 3.5 is split into two rules for choice and tree congruence.

- Rule Chc-Idemp:

```
Theorem cIdemp d e :
  chc d e e <=> e.
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); reflexivity. Qed.
```

- Rule Chc-Chc-Merge where the nested choice is in the left alternative:

```
Theorem ccMergeL d e1 e2 e3 :
  chc d (chc d e1 e2) e3 <=> chc d e1 e3.
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); reflexivity. Qed.
```

- Rule Chc-Chc-Merge where the nested choice is in the right alternative:

```
Theorem ccMergeR d e1 e2 e3 :
  chc d e1 (chc d e2 e3) <=> chc d e1 e3.
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); reflexivity. Qed.
```

- Rule Chc-Chc-Swap where the nested choice is in the left alternative of the simpler form:

```
Theorem ccSwapL d d' e1 e2 e3 :
  chc d' (chc d e1 e3) (chc d e2 e3) <=>
  chc d (chc d' e1 e2) e3.
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); destruct (s d'); reflexivity. Qed.
```

- Rule Chc-Chc-Swap where the nested choice is in the right alternative of the simpler form:

```
Theorem ccSwapR d d' e1 e2 e3 :
  chc d' (chc d e1 e2) (chc d e1 e3) <=>
  chc d e1 (chc d' e2 e3).
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); destruct (s d'); reflexivity. Qed.
```

- Rule Chc-Obj where the choice is nested in the left branch of the tree:

```
Theorem cObjL d a e1 e2 e3 :
  chc d (node a e1 e3) (node a e2 e3) <=>
  node a (chc d e1 e2) e3.
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); reflexivity. Qed.
```

- Rule Chc-Obj where the choice is nested in the right branch of the tree:

```
Theorem cObjR d a e1 e2 e3 :
  chc d (node a e1 e2) (node a e1 e3) <=>
  node a e1 (chc d e2 e3).
Proof.
  unfold equiv. intros s. unfold sem.
  destruct (s d); reflexivity. Qed.
```

- Choice congruence rule:

```
Theorem congChc d e1 e2 e3 e4 :
  e1 <=> e3 / e2 <=> e4 ->
  chc d e1 e2 <=> chc d e3 e4.
Proof.
  unfold equiv. intros H.
  elim H. intros H1 H2 s.
  unfold sem. destruct (s d).
  fold sem. apply H1.
  fold sem. apply H2. Qed.
```

- Tree/object language congruence rule:

```
Theorem congObj a e1 e2 e3 e4 :
  e1 <=> e3 / e2 <=> e4 ->
  node a e1 e2 <=> node a e3 e4.
Proof.
  unfold equiv. intros H.
  elim H. intros H1 H2 s.
  unfold sem. destruct (s d).
  fold sem. apply H1.
  fold sem. apply H2. Qed.
```

- Reflexivity:

```
Theorem equivRefl e :
  e <=> e.
Proof. unfold equiv. reflexivity. Qed.
```

- Symmetry:

```
Theorem equivSymm e e’ :
  e <=> e’ -> e’ <=> e.
Proof.
  unfold equiv. intros H s.
  symmetry. apply H. Qed.
```

- Transitivity:

```
Theorem equivTrans e1 e2 e3 :
  e1 <=> e2 / e2 <=> e3 -> e1 <=> e3.
Proof.
  unfold equiv. intros H.
  elim H. intros H1 H2 s.
  rewrite -> H1. rewrite <- H2. reflexivity. Qed.
```

This represents a complete set of equivalence rules for the choice calculus as defined in Section A.1, determined by enumerating all permutations of the syntactic forms. Since all cases have been proved, the relevant subset of Theorem 3.5.1 is also proved.

Bibliography

D. Abrahams and A. Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. Addison-Wesley Professional, 2004.

M. Adams. Scrap Your Zippers – A Generic Zipper for Heterogeneous Types. In *ACM SIGPLAN Workshop on Generic Programming*, pages 13–24, 2010.

S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Int. Conf. on Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 36–50. Springer-Verlang, 2008b.

S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, Sept. 2010.

D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Int. Software Product Line Conf.*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlang, 2005.

D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. on Software Engineering and Methodology*, 1(4):355–398, 1992.

D. Batory, J. Liu, and J. N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. In *ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, pages 48–57, 2003.

D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, New York, 2004.

E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPLlift - Statically Analyzing Software Product Lines in Minutes Instead of Years. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*, June 2013.

G. Bracha and W. Cook. Mixin-Based Inheritance. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 303–311, 1990.

S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ACM SIGPLAN 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*, 2013. To appear.

A. Classen, A. Hubaux, and P. Heymans. A Formal Semantics for Multi-Level Staged ConïňĄguration. In *Work. on Variability Modeling of Software-Intensive Systems*, pages 51–60, 2009.

A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 321–330, 2011.

M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Trans. on Software Engineering*, 34(5):633–650, 2008.

CPP. The C Preprocessor. http://gcc.gnu.org/onlinedocs/cpp/, accessed: June 2013.

K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Int. Conf. on Generative Programming and Component Engineering*, pages 211–220, 2006.

K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–168, 2005.

L. Damas and R. Milner. Principal Type Schemes for Functional Programming Languages. In *9th ACM Symp. on Principles of Programming Languages*, pages 207–208, 1982.

T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Comm. of the ACM*, 44(10):28–32, 2001.

S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Growing a Language Environment with Editor Libraries. In *Int. Conf. on Generative Programming and Component Engineering*, pages 167–176, 2011a.

S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Growing a Language Environment with Editor Libraries. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 391–406, 2011b.

M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. on Software Engineering*, 28(12):1146–1170, 2002.

M. Erwig and E. Walkingshaw. Program Fields for Continuous Software. In *ACM SIGSOFT Workshop on the Future of Software Engineering Research*, pages 105–108, 2010.

M. Erwig and E. Walkingshaw. Semantics First! Rethinking the Language Design Process. In *Int. Conf. on Software Language Engineering*, volume 6940 of *LNCS*, pages 243–262, 2011a.

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, 2011b.

M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering*, pages 55–99, 2012a.

M. Erwig and E. Walkingshaw. Semantics-Driven DSL Design. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pages 56–80. IGI Global, 2012b.

M. Erwig, K. Ostermann, T. Rendel, and E. Walkingshaw. Adding Configuration to the Choice Calculus. In *Int. Workshop on Variability Modelling of Software-Intensive Systems*, pages 13:1–13:8, 2013a.

M. Erwig, E. Walkingshaw, and S. Chen. An Abstract Representation of Variational Graphs, 2013b. Under review.

J.-M. Favre. The CPP Paradox. In *European Work. on Software Maintenance*, 1995.

J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Int. Conf. on Evaluation and Assessment in Software Engineering*, pages 66–75, 2011.

M. Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, 17(1–3):35–75, 1991.

M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, Cambridge, MA, 2009.

E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas. Evolving Software Product Lines with Aspects. In *IEEE Int. Conf. on Software Engineering*, pages 261–270. IEEE Computer Society, 2008.

M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.

M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages?, 12 June 2005. http://martinfowler.com/articles/languageWorkbench.html.

GHC. The Glasgow Haskell Compiler. http://haskell.org/ghc, accessed: June 2013.

C. A. Gunter. *Semantics of Programming Languages – Structures and Techniques*. MIT Press, Cambridge, MA, 1992.

A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Line Conf.*, 2011.

P. Höfner, R. Khedri, and B. Möller. Feature Algebra. In *Int. Symp. on Formal Methods*, volume 4085 of *LNCS*, pages 300–315. Springer-Verlang, 2006.

P. Hudak. Modular Domain Specific Languages and Tools. In *IEEE Int. Conf. on Software Reuse*, pages 134–142, 1998.

A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

V. Karvonen and P. Mensonides. The Boost Preprocessor Library, 2001. http://boost.org/doc/libs/1_53_0/libs/preprocessor/.

C. Kästner and S. Apel. Integrating Compositional and Annotative Approaches for Product Line Engineering. In *GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, pages 35–40, 2008.

C. Kästner and S. Apel. Virtual Separation of Concerns—A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008a.

C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report FIN-2008-02, University of Magdeburg, Mar. 2008b.

C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Int. Conf. on Generative Programming and Component Engineering*, pages 157–166, 2009a.

C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *Int. Conf. on Objects, Components, Models and Patterns*, volume 33 of *LNBIP*, pages 175–194. Springer-Verlang, 2009b.

C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Int. Software Product Line Conf.*, pages 181–190, 2009c.

C. Kästner, P. G. Giarrusso, and K. Ostermann. Partial Preprocessing C Code for Variability Analysis. In *Work. on Variability Modeling of Software-Intensive Systems*, pages 127–136, 2011a.

C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824, 2011b.

C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 21(3): 14:1–14:39, July 2012a.

C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 773–792, 2012b.

C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Int. Workshop on Feature-Oriented Software Development*, pages 1–8, 2012c.

Kconfig. Kconfig Language Documentation. `http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`, accessed: June 2013.

A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Int. Workshop on Feature-Oriented Software Development*, pages 25–32, 2010.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlang, 1997.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conf. on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–354. Springer-Verlang, 2001.

E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion. In *ACM Conf. on LISP and Functional Programming*, pages 151–161, 1986.

C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

C. W. Krueger. New Methods in Software Product Line Development. In *Software Product Line Conf.*, pages 95–99, 2006.

R. Lämmel and S. Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 26–37, 2003.

R. Lämmel and S. Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *ACM Int. Conf. on Functional Programming*, pages 244–255, 2004.

D. Le, E. Walkingshaw, and M. Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 143–150, 2011.

K. Lee, K. C. Kang, and J. Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *LNCS*, pages 62–77. Springer-Verlang, 2002.

J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 105–114, 2010.

J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Int. Conf. on Aspect-Oriented Software Development*, pages 191–202, 2011.

J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *IEEE Int. Conf. on Software Engineering*, pages 112–121, 2006.

Make. GNU Make. http://www.gnu.org/software/make/manual/make.html, accessed: June 2013.

A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *IEEE Int. Requirements Engineering Conf.*, pages 243–253, 2007.

M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Int. Conf. on Aspect-Oriented Software Development*, pages 90–99, 2003.

M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes*, 29 (6):127–136, 2004.

D. Miller. Abstract Syntax for Variable Binders: An Overview. In *Computational Logic*, LNAI 1861, pages 239–253, 2000.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *Int. Conf. on Software Testing, Verification and Validation*, pages 459–468. IEEE Computer Society, 2010.

B. C. Pierce. Programming with Intersection Types, Union Types, and Polymorphism. Report CMU-CS-91-106, Carnegie Mellon University, 1991.

B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Computer Science Department, Aarhus University Denmark, 1981.

K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg, 2005.

C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conf. on Object-Oriented Programming*, pages 419–443, 1997.

M.-O. Reiser. *Managing Complex Variability in Automotive Software Product Lines with Subscoping and Configuration Links*. PhD thesis, Technische Universität Berlin, Feb. 2009.

M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. *Work. on Variability Modeling of Software-Intensive Systems*, 10:123–130, 2010.

M. Rosenmüller, N. Siegmund, C. Kästner, and S. S. ur Rahman. Modeling Dependent Software Product Lines. In *Work. on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 13–18, 2008.

S. Roychoudhury, J. Gray, H. Wu, J. Zhang, and Y. Lin. A Comparative Analysis of Meta-Programming and Aspect-Orientation. In *ACM Southeast Conf.*, pages 196–201, 2003.

P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *IEEE Int. Requirements Engineering Conf.*, pages 139–148, 2006.

M. L. Scott. *Programming Language Pragmatics*. Elsevier Science, 2009.

S. She and T. Berger. Formal Semantics of the Kconfig Language. Technical report, University of Waterloo, Jan. 2010.

S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The Variability Model of the Linux Kernel. In *Int. Workshop on Variability Modelling of Software-Intensive Systems*, pages 45–51, 2010.

T. Sheard. Accomplishments and Research Challenges in Meta-Programming. In *2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation*, LNCS 2196, pages 2–44, 2001.

H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *Proc. of the USENIX Summer Conf.*, pages 185–198, 1992.

R. Strniša, P. Sewell, and M. Parkinson. The Java Module System: Core Design and Semantic Definition. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 499–514, 2007.

W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *IEEE Int. Conf. on Software Engineering*, pages 107–119, 1999.

R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *GPCE Work. on Feature-Oriented Software Development*, pages 81–86, 2009.

S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Int. Conf. on Generative Programming and Component Engineering*, pages 95–104, 2007.

W. F. Tichy. RCS—A System for Version Control. *Software: Practice and Experience*, 15(7):637–654, 1985.

P. Wadler. Theorems for Free! In *Conf. on Functional Programming and Computer Architecture*, pages 347–359, 1989.

E. Walkingshaw and M. Erwig. A Calculus for Modeling and Implementing Variation. In *Int. Conf. on Generative Programming and Component Engineering*, pages 132–140, 2012.

H. Zhang and S. Jarzabek. XVCL: A Mechanism for Handling Variants in Software Product Lines. *Science of Computer Programming*, 53(3):381–407, 2004.