Formula Choice Calculus

Spencer Hubbard *
Oregon State University, USA
hubbarsp@oregonstate.edu

Eric Walkingshaw
Oregon State University, USA
walkiner@oregonstate.edu

Abstract

The choice calculus is a simple metalanguage and associated theory that has been successfully applied to several problems of interest to the feature-oriented software development community. The formal presentation of the choice calculus essentially restricts variation points, called choices, to vary based on the inclusion or not of a single feature, while in practice variation points may depend on several features. Therefore, in both theoretical applications of the choice calculus, and in tools inspired by the choice calculus, the syntax of choices has often been generalized to depend on an arbitrary propositional formula of features. The purpose of this paper is to put this syntactic generalization on more solid footing by also generalizing the associated theory. Specifically, after defining the syntax and denotational semantics of the formula choice calculus (FCC), we define and prove the soundness of a syntactic equivalence relation on FCC expressions. This effort validates previous work which has implicitly assumed the soundness of rules in the equivalence relation, and also reveals several rules that are specific to FCC. Finally, we describe some further generalizations to FCC and their limits.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Theory

Keywords variation, software product lines, choice calculus, refactoring, program transformation

1. Introduction

The choice calculus [10, 24] is a simple metalanguage for describing variation in programs and other structured data. In the choice calculus, variation is represented in-place as choices between alternative subexpressions. For example, the following variational expression e_{aba} contains three choices.

$$e_{aba} = A\langle 2, 3\rangle + B\langle 4, 5\rangle + A\langle 6, 7 + 8\rangle$$

Each choice has an associated dimension, which is used to synchronize the choice with other choices in different parts of the expression. For example, expression e_{aba} contains two dimensions, A and B, and the two choices in dimension A are synchronized. Therefore, the variational expression e_{aba} represents four different plain expressions, depending on whether the left or right alternatives are selected from each dimension. This is reflected in the semantics of e_{aba}, shown below. The semantics is defined in terms of a *configuration* function c that maps each dimension to either $\mathbf{L}$ or $\mathbf{R}$.[1]

$$[\![e_{aba}]\!]_c = \begin{cases} 2+4+6 & c\,A = \mathbf{L},\ c\,B = \mathbf{L} \\ 2+5+6 & c\,A = \mathbf{L},\ c\,B = \mathbf{R} \\ 3+4+7+8 & c\,A = \mathbf{R},\ c\,B = \mathbf{L} \\ 3+5+7+8 & c\,A = \mathbf{R},\ c\,B = \mathbf{R} \end{cases}$$

The syntax and semantics of the choice calculus are briefly summarized in Section 2. For now, a helpful analogy is to consider the choice calculus as a more concise and constrained version of the C preprocessor's `#ifdef` notation (CPP), where a choice $D\langle e_1, e_2\rangle$ corresponds to the following CPP pattern.

`#ifdef` D `\\` e_1 `\\ #else \\` e_2 `\\ #endif`

Like CPP, the choice calculus is a metalanguage that can be used with different object languages. Unlike CPP, the choice calculus does not operate on plain text, but respects the underlying abstract syntax of the object language. This enforces that in-place variation is "disciplined" [16] and ensures that any variant that can be obtained from a choice calculus expression is syntactically valid in the object language.

The choice calculus is also more restrictive than CPP in that it only allows atomic dimension names as the condition of choices, whereas CPP allows arbitrary propositional formulas in the condition of an `#if` construct. This restriction has drawbacks. For example, suppose we consider each dimension D to represent a feature in a software product line,

* Now at Tableau Software

[1] We use adjacency to indicate function application, as in lambda calculus.

FOSD'16, October 30, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4647-4/16/10...
http://dx.doi.org/10.1145/3001867.3001873

where $c\ D = \mathbf{L}$ means feature D is enabled and $c\ D = \mathbf{R}$ means it is disabled. A common scenario is to replace some existing code e_1 with some new code e_2 to deal with an *interaction* between two features A and B. We can represent this scenario in the choice calculus using nested choices.

$$e_{dup} = A\langle B\langle e_2, e_1\rangle, e_1\rangle$$

But this solution duplicates e_1. In our previous work [10], we introduced a **let**-construct to the choice calculus to enable factoring out such redundancy as **let** v = e_1 **in** $A\langle B\langle e_2, v\rangle, v\rangle$. However, naming substantially increases the complexity of the calculus, so this extension has not been used in most applications of the choice calculus.

An arguably better solution is to generalize choices to allow them to be indexed by propositional formulas of dimensions, allowing us to rewrite e_{dup} as the following expression e_{conj}.

$$e_{conj} = (A \wedge B)\langle e_2, e_1\rangle$$

Given a configuration c, the left alternative e_2 will be selected from e_{conj} only if both $c\ A = \mathbf{L}$ and $c\ B = \mathbf{L}$, otherwise e_1 will be selected. The expression is isomorphic to the following CPP pattern using the more flexible `#if` directive.

`#if` A `&&` B `\\` e_2 `\\ #else \\` e_1 `\\ #endif`

We refer to the choice calculus restricted to atomic dimensions as the *atomic choice calculus* (ACC) and the version generalized to propositional formulas of dimensions as the *formula choice calculus* (FCC). We use "choice calculus" when the distinction does not matter.

There are some reasons to prefer ACC over FCC in formal contexts: it is somewhat simpler and many operations on FCC expressions require solving Boolean satisfiability problems. However, FCC is more expressive and corresponds more closely to CPP, which is very widely used in practice. There is also strong evidence that SAT solvers are fast at solving the kinds of satisfiability problems that arise when working with variational software [17, 20].

Next, observe that the choice calculus representation is not unique—there are many different ways to represent a variational expression with the same semantics. For example, by distributing shared code over the alternatives of a choice, as illustrated below, we can obtain a new expression that is semantically equivalent.

$$A\langle e_1, e_2\rangle + e_3 \ \equiv\ A\langle e_1 + e_3, e_2 + e_3\rangle$$

In our previous work, we have defined and proved sound an equivalence relation consisting of several such syntactic rules relating ACC expressions [10, 24].

This equivalence relation has been very useful in practice. For example, it the basis for a variational type unification algorithm [5, 6] that has applications to typing software product lines, C++ templates, generalized algebraic data types, and improving type error reporting [2–4]. This equivalence relation has also been used to define the update operation of a view-based editing model for variational software [23, 25], and is the (usually implicit) theoretical basis for languages and tools for variational programming [7, 11, 14, 19, 21, 26].

The major contribution of this work is to extend ACC*'s equivalence relation to* FCC. In particular, in Section 3, we formally define the syntax and semantics of FCC, then in Section 4, we enumerate a set of rules that syntactically relate semantically equivalent expressions. Some of these rules are straightforward generalizations of corresponding ACC rules, while some are new and specific to FCC. We provide human-readable soundness proofs for a representative set of rules in the text, and also a complete mechanized soundness proof of all the rules using the Coq proof assistant [1].

This effort is significant since it validates existing work that uses FCC and was already relying on the correctness of the corresponding rules from ACC [23, 25, 26]. Additionally, many other languages and tools that use FCC or very similar representations, such as TypeChef [13, 15], can benefit by using this relation as a sound basis for defining semantic-preserving transformations and optimizations.

In Section 5, we discuss further generalizations of FCC to other Boolean algebras. We discuss related work in Section 6 and future work and conclusions in Section 7.

2. Atomic Choice Calculus

In this section, we briefly present ACC as necessary background for the development of FCC in Section 3. The language presented here is simpler than the original presentation of the choice calculus [10] since it omits local dimension declarations and the **let**-construct described in Section 1. However, this simplified ACC has been used in the majority of applications and its corresponding equivalence relation has also been proved sound [24].

The syntax and semantics of ACC are defined in Figure 1. The choice calculus is a metalanguage that can be instantiated by an arbitrary object language. Here we represent the abstract syntax of the object language X as a labeled binary tree. This choice is made for simplicity, but is not a fundamental restriction. Note that a binary tree supports encoding internal nodes of arbitrary arity, for example, with right-nested trees. The syntax of ACC is then just X extended by dimension-labeled choices.

A *configuration* is a (total) function from dimensions to *tags* that describes how to configure an expression in ACC into one of its variant plain programs in X. The tag $\mathbf{L}$ means to select the left alternative of every choice in a given dimension, while $\mathbf{R}$ means to select the right alternatives. The metavariables t and c are used to represent arbitrary tags and configurations, respectively, unless qualified otherwise.

The semantic domain for ACC expressions is the function domain $C \to X$, which is the set of functions from configurations to terms in the object language. The semantic

Generic object language syntax:

$$\begin{array}{lcll} a \in A & & & \textit{AST Label} \\ x \in X & ::= & \varepsilon & \textit{AST Leaf} \\ & | & a\prec x, x\succ & \textit{AST Node} \end{array}$$

Atomic choice calculus syntax:

$$\begin{array}{lcll} d \in D & & & \textit{Dimension} \\ e \in E & ::= & \varepsilon & \textit{AST Leaf} \\ & | & a\prec e, e\succ & \textit{AST Node} \\ & | & d\langle e, e\rangle & \textit{Choice} \end{array}$$

Tags and configurations:

$$\begin{array}{lcll} t \in T & ::= & \mathbf{L} \mid \mathbf{R} & \textit{Tag} \\ c \in C & = & D \to T & \textit{Configuration} \end{array}$$

Semantics of atomic choice calculus:

$$\begin{aligned} &E[\![\cdot]\!] : E \to C \to X \\ &E[\![\varepsilon]\!]_c = \varepsilon \\ &E[\![a\prec e_1, e_2\succ]\!]_c = a\prec E[\![e_1]\!]_c, E[\![e_2]\!]_c\succ \\ &E[\![d\langle e_1, e_2\rangle]\!]_c = \begin{cases} E[\![e_1]\!]_c, & \text{if } c\ d = \mathbf{L} \\ E[\![e_2]\!]_c, & \text{otherwise} \end{cases} \end{aligned}$$

Figure 1: Atomic choice calculus language definition.

function $E[\![\cdot]\!]$ is defined in Figure 1. In the last equation, note that if $c\ d \neq \mathbf{L}$, then $c\ d = \mathbf{R}$. Elements of the object language in the image of $E[\![e]\!]$ are called *variants* of e and application of $E[\![e]\!]$ to a given c is called *configuration*.

To illustrate both the semantics and how binary trees can encode other object languages, consider the choice calculus instantiated with the object language of decimal notation. Labels in this object language are Arabic numerals, such as 1, 2, and 3. Terms of this object language are right nested branches. For convenience, we write the concrete syntax of terms as sequences of Arabic numerals, for example, 123 and 213 are terms in concrete syntax which correspond to the terms $1\prec\varepsilon, 2\prec\varepsilon, 3\prec\varepsilon, \varepsilon\succ\succ$ and $2\prec\varepsilon, 1\prec\varepsilon, 3\prec\varepsilon, \varepsilon\succ\succ\succ$, respectively, in the abstract syntax. Suppose A is a dimension. Then $1\,A\langle 23, 32\rangle$ is a choice calculus expression. This expression consists of a node labeled 1 with an empty left child and a choice as the right child. The choice is labeled with the dimension A and the alternatives are the terms 23 and 32. For each configuration c, the semantics of this expression is the following: (1) if $c\ d = \mathbf{L}$, then the variant 123 is selected, and (2) if $c\ d = \mathbf{R}$, then the variant 132 is selected.

It is easy to extend the definition of ACC to support choices with more alternatives. For example, to support three-alternative choices, add a new element to the set of tags, add an additional alternative to the syntax of choices, and extend the semantic function to include a new case in the semantics of choices.

3. Formula Choice Calculus

In this section we present FCC. As described in the introduction, this language has already been widely used. The presentation in this paper is similar to one given in previous work [25], but more thorough. First, we define propositional formulas of tags and dimensions in Section 3.1, then define the syntax and semantics of FCC in Section 3.2.

3.1 Propositional Formulas

We begin with a "semantics first" [9, 12] description of propositional formulas over tags. The set T forms an algebra with a prefix unary operator ($\neg$), called *complement*, and infix binary operators ($\vee$) and ($\wedge$), called *join* and *meet*, respectively. These operators are defined by the tables below.

t	$\neg t$
$\mathbf{L}$	$\mathbf{R}$
$\mathbf{R}$	$\mathbf{L}$

$\vee$	$\mathbf{L}$	$\mathbf{R}$
$\mathbf{L}$	$\mathbf{L}$	$\mathbf{L}$
$\mathbf{R}$	$\mathbf{L}$	$\mathbf{R}$

$\wedge$	$\mathbf{L}$	$\mathbf{R}$
$\mathbf{L}$	$\mathbf{L}$	$\mathbf{R}$
$\mathbf{R}$	$\mathbf{R}$	$\mathbf{R}$

Note that T is just the Boolean algebra with two elements, up to isomorphism, where $\mathbf{L}$ is "true" and $\mathbf{R}$ is "false".

The syntax and semantics of formulas over tags and dimensions is given in the first half of Figure 2. Note that for convenience we reuse in the syntax the semantic-level operators ($\neg, \vee, \wedge$) defined by the tables above (and used in the right-hand side of the semantics definition). The semantic domain for formulas is the function domain $C \to T$. In other words, given a formula (e.g. attached to a choice), the semantics tells us whether to select the left alternative or the right for a given configuration. The semantic function for formulas is $F[\![\cdot]\!]$, defined by the equations in Figure 2. This is isomorphic to the evaluation of Boolean expressions with an environment (configuration) mapping variables (dimensions) to truth values (tags).

3.2 Language Definition

The syntax and semantics of FCC are defined in the second half of Figure 2. Note that although we repurpose the meta-variable e and sort E to FCC (and these refer to FCC terms hereafter), we directly reuse the sort X to refer to plain object language terms, as defined in Section 2.

The syntax of FCC is the same as ACC, except we label choices by formulas rather than atomic dimensions. The semantic domain of FCC remains unchanged, $C \to X$, mapping configurations to terms in the object language. The semantic function differs in the case for a choice $f\langle e_1, e_2\rangle$, which now the configuration to evaluate the formula f rather than simply looking up the tag associated with the dimension.

To illustrate, consider FCC instantiated by the object language of decimal notation, as before. Suppose A and B are dimensions. Then $1(A \vee B)\langle 23, 32\rangle$ is an FCC expression.

Formula syntax:

$$\begin{array}{rcll} f \in F & ::= & t & \textit{Tag} \\ & | & d & \textit{Dimension} \\ & | & \neg\, f & \textit{Complement} \\ & | & f \vee f & \textit{Join} \\ & | & f \wedge f & \textit{Meet} \end{array}$$

Semantics of formulas:

$$\begin{aligned} &F[\![\cdot]\!] : F \to C \to T \\ &F[\![t]\!]_c &&= t \\ &F[\![d]\!]_c &&= c\, d \\ &F[\![\neg\, f]\!]_c &&= \neg\, F[\![f]\!]_c \\ &F[\![f_1 \vee f_2]\!]_c &&= F[\![f_1]\!]_c \vee F[\![f_2]\!]_c \\ &F[\![f_1 \wedge f_2]\!]_c &&= F[\![f_1]\!]_c \wedge F[\![f_2]\!]_c \end{aligned}$$

Formula choice calculus syntax:

$$\begin{array}{rcll} e \in E & ::= & \varepsilon & \textit{Empty} \\ & | & a{\prec}e, e{\succ} & \textit{Tree} \\ & | & f\langle e, e\rangle & \textit{Choice} \end{array}$$

Semantics of formula choice calculus:

$$\begin{aligned} &E[\![\cdot]\!] : E \to C \to X \\ &E[\![\varepsilon]\!]_c &&= \varepsilon \\ &E[\![a{\prec}e_1, e_2{\succ}]\!]_c &&= a{\prec}E[\![e_1]\!]_c, E[\![e_2]\!]_c{\succ} \\ &E[\![f\langle e_1, e_2\rangle]\!]_c &&= \begin{cases} E[\![e_1]\!]_c, & \text{if } F[\![f]\!]_c = \mathbf{L} \\ E[\![e_2]\!]_c, & \text{otherwise} \end{cases} \end{aligned}$$

Figure 2: Formula choice calculus language definition.

Now, for each configuration c, the semantics of this expression is: (1) if $c\ A = \mathbf{L}$ or $c\ B = \mathbf{L}$, then the variant 123 is selected; (2) if $c\ A = \mathbf{R}$ and $c\ B = \mathbf{R}$, then the variant 132 is selected.

4. Equivalence Relation

In this section we define and prove sound a syntactic relation ($\equiv$) on FCC terms, where if two terms $e_1 \equiv e_2$ are related, then e_1 and e_2 are semantically equivalent, that is, $E[\![e_1]\!] = E[\![e_2]\!]$. The rules that make up this equivalence relation can therefore be used to prove the semantic equivalence of two FCC expressions, or if applied in a directed fashion, can be used to transform an FCC expression in a semantics-preserving way.

We begin in Section 4.1 by defining an equivalence relation on formulas, then define the equivalence relation on FCC in Section 4.2. We provide human-friendly proofs for an illustrative subset of the rules directly in the text. A mechanized proof of the complete equivalence relation, written in Coq [1], is available online.[2]

4.1 Formula Equivalence

In this subsection, we establish syntactic rules for deriving the semantic equivalence of formulas. This relation is needed in the premises of several rules of the equivalence relation for FCC defined in Section 4.2. However, in practice, formula equivalence can be checked using a SAT solver.

Before we proceed to the definition, it is useful to recall the following elementary facts:

1. Two functions are equal, by definition, if they have the same domain and codomain and their images agree for all elements in the domain.
2. For any function, the inverse images of elements in the codomain partition the domain.
3. Any partition defines a "canonical" equivalence relation where the parts of the partition correspond to the equivalence classes of the relation.

Definition 4.1 (Formula equivalence)**.** Let ($\equiv$) be a binary relation on formulas defined by $f \equiv f'$ iff $F[\![f]\!] = F[\![f']\!]$.

Note that formula equivalence is defined in terms of function equality. Since $F[\![\cdot]\!]$ is a well-defined function from formulas to elements of the semantic domain, it follows from earlier remarks that the relation ($\equiv$) is an equivalence relation. We refer to the relation ($\equiv$) as *semantic equivalence* and we call formulas f and f' (*semantically*) *equivalent* if $f \equiv f'$.

A set of formula equivalence rules are given in Figure 3. They are organized into three groups. The structural rules state that formula equivalence is reflexive, symmetric, and transitive. Reflexivity and symmetry follow directly from Definition 4.1, and transitivity follows from Definition 4.1 and the definition of $F[\![\cdot]\!]$.

The congruence rules state that two formulas with the same top-level connective are semantically equivalent if their subexpressions are equivalent. Note that these rules are not independent of each other. For example, the JOIN-CONG rule can be derived from JOIN-CONG-L and JOIN-CONG-R. Alternatively, both JOIN-CONG-L and JOIN-CONG-R can be derived from JOIN-CONG and REFL.

Finally, the algebraic rules given in Figure 3 encode several of the usual laws of Boolean algebra expressed in terms of formulas. These rules include axioms that characterize the behavior of complementation (JOIN-COMP and MEET-COMP) and identities for join and meet (MEET-ID and JOIN-ID); rules that describe the idempotence, associativity, and commutativity of join and meet; and rules for distributing join and meet over each other. The COMP-JOIN and COMP-MEET rules are the De Morgan's laws for formulas. Since these rules are all standard, we include only a subset here. However, a more complete set (including annihilators for join and meet,

[2] `https://github.com/lambda-land/FCC-Coq`

Structural rules:

$$\textsc{Refl-F}\ \ f \equiv f \qquad \textsc{Symm-F}\ \ \frac{f \equiv f'}{f' \equiv f} \qquad \textsc{Tran-F}\ \ \frac{f_1 \equiv f_2 \qquad f_2 \equiv f_3}{f_1 \equiv f_3}$$

Congruence rules:

$$\textsc{Comp-Cong}\ \ \frac{f \equiv f'}{\neg f \equiv \neg f'} \qquad \textsc{Join-Cong}\ \ \frac{f_1 \equiv f_1' \qquad f_2 \equiv f_2'}{f_1 \vee f_2 \equiv f_1' \vee f_2'} \qquad \textsc{Meet-Cong}\ \ \frac{f_1 \equiv f_1' \qquad f_2 \equiv f_2'}{f_1 \wedge f_2 \equiv f_1' \wedge f_2'}$$

$$\textsc{Join-Cong-L}\ \ \frac{f_1 \equiv f_1'}{f_1 \vee f_2 \equiv f_1' \vee f_2} \qquad \textsc{Join-Cong-R}\ \ \frac{f_2 \equiv f_2'}{f_1 \vee f_2 \equiv f_1 \vee f_2'}$$

$$\textsc{Meet-Cong-L}\ \ \frac{f_1 \equiv f_1'}{f_1 \wedge f_2 \equiv f_1' \wedge f_2} \qquad \textsc{Meet-Cong-R}\ \ \frac{f_2 \equiv f_2'}{f_1 \wedge f_2 \equiv f_1 \wedge f_2'}$$

Algebraic rules (subset):

$$\textsc{Join-Comp}\ \ f \vee \neg f \equiv \mathbf{L} \qquad \textsc{Meet-Comp}\ \ f \wedge \neg f \equiv \mathbf{R} \qquad \textsc{Join-Id}\ \ \mathbf{R} \vee f \equiv f \qquad \textsc{Meet-Id}\ \ \mathbf{L} \wedge f \equiv f$$

$$\textsc{Join-Idemp}\ \ f \vee f \equiv f \qquad \textsc{Join-Assoc}\ \ f_1 \vee (f_2 \vee f_3) \equiv (f_1 \vee f_2) \vee f_3$$

$$\textsc{Join-Comm}\ \ f_1 \vee f_2 \equiv f_2 \vee f_1 \qquad \textsc{Join-Dist}\ \ f_1 \vee (f_2 \wedge f_3) \equiv (f_1 \vee f_2) \wedge (f_1 \vee f_3)$$

$$\textsc{Meet-Idemp}\ \ f \wedge f \equiv f \qquad \textsc{Meet-Assoc}\ \ f_1 \wedge (f_2 \wedge f_3) \equiv (f_1 \wedge f_2) \wedge f_3$$

$$\textsc{Meet-Comm}\ \ f_1 \wedge f_2 \equiv f_2 \wedge f_1 \qquad \textsc{Meet-Dist}\ \ f_1 \wedge (f_2 \vee f_3) \equiv (f_1 \wedge f_2) \vee (f_1 \wedge f_3)$$

$$\textsc{Comp-Join}\ \ \neg(f_1 \vee f_2) \equiv \neg f_1 \wedge \neg f_2 \qquad \textsc{Comp-Meet}\ \ \neg(f_1 \wedge f_2) \equiv \neg f_1 \vee \neg f_2$$

Figure 3: Formula equivalence relation.

double negation, symmetric identity laws, etc.) is defined and proved sound with respect to Definition 4.1 in the online Coq implementation.

Clearly, the formula equivalence relation implies that the set of equivalence classes of formulas is isomorphic to the free Boolean algebra on the set of dimensions D. The operations on equivalence classes are defined in terms of the operations on representative formulas and the formula congruence rules ensure that these operations are well-defined.

As an aside, consider the binary infix operator ($\triangle$) on the set of equivalence classes, called *symmetric difference* (or

Structural rules:

$$\textsc{Refl-E}\ \ e \equiv e \qquad \textsc{Symm-E}\ \ \frac{e \equiv e'}{e' \equiv e} \qquad \textsc{Tran-E}\ \ \frac{e_1 \equiv e_2 \qquad e_2 \equiv e_3}{e_1 \equiv e_3}$$

Choice transposition and congruence rules:

$$\textsc{Chc-Tran}\ \ f\langle e_1, e_2\rangle \equiv \neg f\langle e_2, e_1\rangle \qquad \textsc{Obj-Cong}\ \ \frac{e_1 \equiv e_1' \qquad e_2 \equiv e_2'}{a\prec e_1, e_2\succ \equiv a\prec e_1', e_2'\succ}$$

$$\textsc{Obj-Cong-L}\ \ \frac{e_1 \equiv e_1'}{a\prec e_1, e_2\succ \equiv a\prec e_1', e_2\succ} \qquad \textsc{Obj-Cong-R}\ \ \frac{e_2 \equiv e_2'}{a\prec e_1, e_2\succ \equiv a\prec e_1, e_2'\succ}$$

$$\textsc{Chc-Cong}\ \ \frac{e_1 \equiv e_1' \qquad e_2 \equiv e_2' \qquad f \equiv f'}{f\langle e_1, e_2\rangle \equiv f'\langle e_1', e_2'\rangle} \qquad \textsc{Chc-Cong-F}\ \ \frac{f \equiv f'}{f\langle e_1, e_2\rangle \equiv f'\langle e_1, e_2\rangle}$$

$$\textsc{Chc-Cong-L}\ \ \frac{e_1 \equiv e_1'}{f\langle e_1, e_2\rangle \equiv f\langle e_1', e_2\rangle} \qquad \textsc{Chc-Cong-R}\ \ \frac{e_2 \equiv e_2'}{f\langle e_1, e_2\rangle \equiv f\langle e_1, e_2'\rangle}$$

Figure 4: Structural, transposition, and congruence rules.

exclusive or). For any representative formulas $f_1, f_2 \in F$, the symmetric difference $f_1 \triangle f_2$ is defined to be the equivalence class with representative formula $(f_1 \wedge \neg f_2) \vee (f_2 \wedge \neg f_1)$. Note that the equivalence classes form a group with the law of composition given by symmetric difference. We revisit this idea in Section 5.

4.2 FCC Term Equivalence

In this subsection, we establish syntactic rules for deriving the semantic equivalence of FCC terms, which is the main contribution of this work. The equivalence of FCC terms is defined in the same way as it is for formulas.

Definition 4.2 (FCC term equivalence)**.** Let ($\equiv$) be a binary relation on expressions defined by $e \equiv e'$ iff $E[\![e]\!] = E[\![e']\!]$.

By the same reasoning as for formula equivalence, the relation ($\equiv$) is an equivalence relation; we say that e and e' are (semantically) equivalent if $e \equiv e'$.

Each of the equivalence rules stated in this section can be proved sound directly from Definition 4.2 and the definition of the semantic function, $E[\![\cdot]\!]$. However, such proofs are not very insightful and also duplicate a lot of logic among the proofs. To avoid duplication and to highlight relationships among rules, we prove new rules by derivation using previous rules, whenever possible. In the Coq implementation, we often provide both proofs—one directly from the definition, and one by derivation with other rules.

The first set of rules are given in Figure 4. Neither the structural rules nor the object congruence rules refer to formula choices, so they are identical to the corresponding rules in ACC from previous work [24]. The structural rules follow directly from the definitions. For the object congruence rules, one can prove OBJ-CONG directly and then derive OBJ-CONG-L and OBJ-CONG-R from OBJ-CONG and REFL (or derive the first after proving the latter two directly).

The choice transposition rule, CHC-TRAN, states that the semantics of a choice is invariant under transposition of its alternatives and complementation of its formula. We prove this directly from the definitions. The CHC-TRAN rule is extremely useful in derivations. Since it is simple and appears so frequently, we often use this rule implicitly.

Like the object congruence rules, the choice congruence rules are not independent. This is illustrated in the proof of the following theorem.

Theorem 4.3. The choice congruence rules hold for all formulas $f, f' \in F$ and expressions $e_1, e_1', e_2, e_2' \in E$.

Proof. First, we prove CHC-CONG-F and CHC-CONG-L directly from Definition 4.1. Next, we show CHC-CONG-R by deriving the consequent from the antecedent using CHC-CONG-L. The derivation is as follows.

$$f\langle e_1, e_2\rangle \;\equiv\; \neg\, f\langle e_2, e_1\rangle \;\equiv\; \neg\, f\langle e_2', e_1\rangle \;\equiv\; f\langle e_1, e_2'\rangle$$

Finally, CHC-CONG follows from the other three rules. □

Observe that converses of the congruence rules for choices do not hold. That is, in general $f\langle e_1, e_2\rangle \equiv f'\langle e_1, e_2\rangle$ is not sufficient to conclude that $f \equiv f'$, and $f\langle e_1, e_2\rangle \equiv f\langle e_1', e_2\rangle$ is not sufficient to conclude $e_1 \equiv e_1'$. For a counterexample to the converse of CHC-CONG-F, consider $\mathbf{L}\langle a, a\rangle \equiv a \equiv \mathbf{R}\langle a, a\rangle$ but $\mathbf{L} \not\equiv \mathbf{R}$. For a counterexample to the converse of CHC-CONG-L, consider $\mathbf{R}\langle a, a\rangle \equiv a \equiv \mathbf{R}\langle a', a\rangle$ but $a \not\equiv a'$ if $a \neq a'$. A similar counterexample can be constructed to disprove the converse of CHC-CONG-R.

Throughout the rest of this section, we use the structural and congruence rules implicitly.

The next set of rules is given in Figure 5. When viewed as semantics preserving transformations, applied left-to-right, these rules can be used to simplify FCC expressions and merge choices that contain identical or redundant alternatives. The choice simplification rules describe cases where the semantics of a choice is equivalent to one of its alternatives.

Theorem 4.4. The choice simplification rules hold for all formulas $f \in F$ and expressions $e, e_1, e_2 \in E$.

Proof. First, we prove CHC-IDEMP and CHC-L directly from Definition 4.2. Next, we derive CHC-R from CHC-L using CHC-TRAN and the fact $\neg\, \mathbf{R} \equiv \mathbf{L}$, which can be shown from the algebraic rules of formula equivalence (see Section 4.1).

The derivation of CHC-R is as follows.

$$\mathbf{R}\langle e_1, e_2\rangle \;\equiv\; \neg\, \mathbf{R}\langle e_2, e_1\rangle \;\equiv\; \mathbf{L}\langle e_2, e_1\rangle \;\equiv\; e_2 \qquad \square$$

Choice simplification rules:

CHC-IDEMP	CHC-L	CHC-R
$f\langle e, e\rangle \equiv e$	$\mathbf{L}\langle e_1, e_2\rangle \equiv e_1$	$\mathbf{R}\langle e_1, e_2\rangle \equiv e_2$

Formula choice merge rules:

CHC-JOIN
$f_1\langle e_1, f_2\langle e_1, e_2\rangle\rangle \equiv (f_1 \vee f_2)\langle e_1, e_2\rangle$

CHC-MEET
$f_1\langle f_2\langle e_1, e_2\rangle, e_2\rangle \equiv (f_1 \wedge f_2)\langle e_1, e_2\rangle$

CHC-JOIN-COMP
$f_1\langle e_1, f_2\langle e_2, e_1\rangle\rangle \equiv (f_1 \vee \neg\, f_2)\langle e_1, e_2\rangle$

CHC-MEET-COMP
$f_1\langle f_2\langle e_2, e_1\rangle, e_2\rangle \equiv (f_1 \wedge \neg\, f_2)\langle e_1, e_2\rangle$

Generic choice merge rules:

CC-MERGE
$f\langle f\langle e_1, e_2\rangle, f\langle e_3, e_4\rangle\rangle \equiv f\langle e_1, e_4\rangle$

CC-MERGE-L	CC-MERGE-R
$f\langle f\langle e_1, e_2\rangle, e_3\rangle \equiv f\langle e_1, e_3\rangle$	$f\langle e_1, f\langle e_2, e_3\rangle\rangle \equiv f\langle e_1, e_3\rangle$

Figure 5: Choice simplification and merge rules.

When viewed as a left-to-right transformation, the formula choice merge rules reveal situations where redundant alternatives in nested choices can be eliminated by combining the formulas of the inner and outer choices.

Theorem 4.5. The formula choice merge rules hold for all formulas $f_1, f_2 \in F$ and expressions $e_1, e_2 \in E$.

Proof. First, we prove CHC-JOIN directly from Definition 4.2. Next, we show CHC-MEET by CHC-JOIN and COMP-MEET. The derivation of CHC-MEET is as follows.

$$\begin{aligned} f_1\langle f_2\langle e_1, e_2\rangle, e_2\rangle &\equiv \neg\, f_1\langle e_2, \neg\, f_2\langle e_2, e_1\rangle\rangle \\ &\equiv (\neg\, f_1 \vee \neg\, f_2)\langle e_2, e_1\rangle \\ &\equiv \neg(f_1 \wedge f_2)\langle e_2, e_1\rangle \\ &\equiv (f_1 \wedge f_2)\langle e_1, e_2\rangle \end{aligned}$$

Finally, we show CHC-JOIN-COMP and CHC-MEET-COMP by CHC-JOIN and CHC-MEET, respectively. The derivation of CHC-JOIN-COMP is as follows.

$$\begin{aligned} f_1\langle e_1, f_2\langle e_2, e_1\rangle\rangle &\equiv f_1\langle e_1, \neg\, f_2\langle e_1, e_2\rangle\rangle \\ &\equiv (f_1 \vee \neg\, f_2)\langle e_1, e_2\rangle \end{aligned}$$

Choice-object commutation rule:

CO-SWAP
$f\langle a\prec e_1, e_2\succ, a\prec e_3, e_4\succ\rangle \equiv a\prec f\langle e_1, e_3\rangle, f\langle e_2, e_4\rangle\succ$

Choice-choice commutation rules:

CC-SWAP
$f_1\langle f_2\langle e_1, e_2\rangle, f_2\langle e_3, e_4\rangle\rangle \equiv f_2\langle f_1\langle e_1, e_3\rangle, f_1\langle e_2, e_4\rangle\rangle$

CC-SWAP-L
$f_1\langle f_2\langle e_1, e_2\rangle, e_3\rangle \equiv f_2\langle f_1\langle e_1, e_3\rangle, f_1\langle e_2, e_3\rangle\rangle$

CC-SWAP-R
$f_1\langle e_1, f_2\langle e_2, e_3\rangle\rangle \equiv f_2\langle f_1\langle e_1, e_2\rangle, f_1\langle e_1, e_3\rangle\rangle$

Figure 6: Choice commutation rules.

And the derivation of CHC-MEET-COMP is as follows.

$$\begin{aligned} f_1\langle f_2\langle e_2, e_1\rangle, e_2\rangle &\equiv f_1\langle \neg\, f_2\langle e_1, e_2\rangle, e_2\rangle \\ &\equiv (f_1 \wedge \neg\, f_2)\langle e_1, e_2\rangle \end{aligned}$$ □

The generic choice merge rules are directly adapted from ACC, and capture the idea that outer choices dominate inner choices labeled by the same formula. The left-to-right application of these rules can be used to eliminate unreachable alternatives in nested choices. For example, consider the expression $f\langle f\langle e_1, e_2\rangle, e_3\rangle$, and observe that if f evaluates to $\mathbf{L}$ for a given configuration c, then the first alternative of both the inner and outer choices will be selected, yielding e_1, otherwise e_3 will be selected. The alternative e_2 is unreachable and so can be safely eliminated by applying the rule CC-MERGE-L.

Theorem 4.6. The generic choice merge rules hold for all formulas $f_1, f_2 \in F$, and expressions $e_1, e_2, e_3, e_4 \in E$.

Proof. First, we prove CC-MERGE-L directly from Definition 4.2. Next, we show CC-MERGE-R by CC-MERGE-L. The derivation of CC-MERGE-R is as follows.

$$\begin{aligned} f\langle e_1, f\langle e_2, e_3\rangle\rangle &\equiv \neg\, f\langle \neg\, f\langle e_3, e_2\rangle, e_1\rangle \\ &\equiv \neg\, f\langle e_3, e_1\rangle \\ &\equiv f\langle e_1, e_3\rangle \end{aligned}$$

CC-MERGE follows directly from the other two rules. □

The last set of rules are the choice commutation rules, stated in Figure 6. The rule CO-SWAP describes the commutation of choices with internal nodes in the AST ("O" stands for object language, and is used for consistency with previous work). This rule is the same as in ACC and can be proved directly from Definition 4.2. The CO-SWAP rule is significant because it can be used (applied right-to-left) to factor out commonalities among the alternatives of a choice, and so is essential to minimizing the representation of a variational expression. The rule is also useful (applied left-to-right) to increase the granularity of choices, which is often needed, for example, during variational execution [7].

In this paper, the CO-SWAP rule and the object congruence rules are defined in terms of binary trees. However, corresponding rules can be easily generated for whatever object language the choice calculus is instantiated by [24].

Finally, the choice-choice commutation rules describe the commutation of choices with each other. These are also adopted directly from ACC.

Theorem 4.7. The choice-choice commutation rules hold for all formulas $f_1, f_2 \in F$, and expressions $e_1, e_2, e_3, e_4 \in E$.

Proof. First, we prove CC-SWAP directly from Definition 4.2. Next, we show CC-SWAP-L by applying CHC-IDEMP then CC-SWAP, producing the following derivation.

$$\begin{aligned} f_1\langle f_2\langle e_1, e_2\rangle, e_3\rangle &\equiv f_1\langle f_2\langle e_1, e_2\rangle, f_2\langle e_3, e_3\rangle\rangle \\ &\equiv f_2\langle f_1\langle e_1, e_3\rangle, f_1\langle e_2, e_3\rangle\rangle \end{aligned}$$

Finally, we show CC-SWAP-R by CC-SWAP-L. The derivation of CC-SWAP-R is as follows.

$$\begin{aligned} f_1\langle e_1, f_2\langle e_2, e_3\rangle\rangle &\equiv \neg\, f_1\langle f_2\langle e_2, e_3\rangle, e_1\rangle \\ &\equiv f_2\langle \neg\, f_1\langle e_2, e_1\rangle, \neg\, f_1\langle e_3, e_1\rangle\rangle \\ &\equiv f_2\langle f_1\langle e_1, e_2\rangle, f_1\langle e_1, e_3\rangle\rangle \end{aligned}$$ □

5. Further Generalizations

At the end of Section 2, we described how to extend ACC to support choices with more than two alternatives by extending the set of tags, adding an alternative to each choice, and updating the corresponding case of the semantic function. It is also straightforward to update the equivalence relation by systematically updating all rules that involve choices. In fact, in the original presentation of the equivalence relation for ACC, we expressed the rules in a way that allowed for choices of arbitrary arity [10].

While it might seem that one could generalize FCC and its equivalence relation in the same way, it turns out this is not the case. In particular, the soundness of any rule that uses a connective in a formula depends on the assumption that the set of tags forms a Boolean algebra. Since any non-trivial Boolean algebra has an even number of elements, it follows that FCC cannot be extended to incorporate choices with an odd number of alternatives. At least, not without sacrificing many of the equivalence rules.

To see why any non-trivial Boolean algebra has an even number of elements, note that any Boolean algebra is a group with law of composition given by symmetric difference. In such a group, every element is its own inverse. This means that the order of a non-identity element is two, which means

t	$\neg t$
$\mathbf{1}$	$\mathbf{4}$
$\mathbf{2}$	$\mathbf{3}$
$\mathbf{3}$	$\mathbf{2}$
$\mathbf{4}$	$\mathbf{1}$

$\vee$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$	$\mathbf{4}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{1}$
$\mathbf{2}$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{1}$	$\mathbf{2}$
$\mathbf{3}$	$\mathbf{1}$	$\mathbf{1}$	$\mathbf{3}$	$\mathbf{3}$
$\mathbf{4}$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$	$\mathbf{4}$

$\wedge$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$	$\mathbf{4}$
$\mathbf{1}$	$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$	$\mathbf{4}$
$\mathbf{2}$	$\mathbf{2}$	$\mathbf{2}$	$\mathbf{4}$	$\mathbf{4}$
$\mathbf{3}$	$\mathbf{3}$	$\mathbf{4}$	$\mathbf{3}$	$\mathbf{4}$
$\mathbf{4}$	$\mathbf{4}$	$\mathbf{4}$	$\mathbf{4}$	$\mathbf{4}$

Figure 7: Boolean algebra on four tags.

the order of the group is a multiple of two by Lagrange's theorem [8, p. 89].

Notwithstanding this limitation, choices in FCC can be extended with an even number of alternatives. Consider the following example of extending choices with four alternatives. The set $T = \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}\}$ together with the operators defined by the tables in Figure 7 is the Boolean algebra with four elements, up to isomorphism, where $\mathbf{1}$ is "true" and $\mathbf{4}$ is "false". The syntax and semantics of formulas is defined in the same way as before. For expressions, we extend the syntax and semantics of choices in a similar way as for ACC.
The equivalence rules for choices must be updated as well, but now this is a systematic adaptation of the existing rules, using the Boolean algebra in Figure 7 as a guide. For example, the choice transposition rule CHC-TRAN is replaced by the following rule.

$$\neg\, f\langle e_1, e_2, e_3, e_4\rangle \equiv f\langle e_4, e_3, e_2, e_1\rangle$$

And the revised CHC-MEET rule looks as follows.

$$\begin{aligned} &f_1\langle f_2\langle e_1, e_2, e_3, e_4\rangle,\\ &\quad f_2\langle e_2, e_2, e_4, e_4\rangle,\\ &\quad f_2\langle e_3, e_4, e_3, e_4\rangle, e_4\rangle \equiv (f_1 \wedge f_2)\langle e_1, e_2, e_3, e_4\rangle \end{aligned}$$

We leave open the question of whether higher-arity formula choice calculuses are useful in practice.

6. Related Work

In addition to CPP's `#if` construct, which is ubiquitous in C and C++ software, many research tools use variation representations quite similar to FCC, and can therefore benefit from the explication provided by this paper, and from new semantics-preserving transformations suggested by the equivalence relation. Kästner et al.'s TypeChef tool [13, 15] includes a representation of formula-based choices that has been reused in a wide variety of projects including variability-aware type and data-flow analyses [17], a variability-aware refactoring engine [18], and several variational interpreters [14, 19, 21].

A concise definition of FCC was originally defined for use in a view-based editing model for variational software [23, 25]. In that work, we discussed the generalization of equivalence rules from ACC and identified some of the FCC-specific rules presented in Section 4. However, the previous work presented the new rules without proof of their correctness, and also omitted some basic rules (CHC-TRAN, CHC-L, and CHC-R) that are useful in many derivations.

In previous work on variational data structures [26], we have discussed the trade-off between variation encodings that use atomic dimensions vs. Boolean formulas. These tradeoffs apply directly when comparing ACC to FCC. The main disadvantages of atomic dimensions are: (1) common variation patterns (e.g. include code to deal with a feature interaction) lead to redundancy in the representation or else require an explicit reuse mechanism that complicates the language, and (2) atomic dimensions do not align well with representations like CPP that are widely used in practice, which is significant if using the choice calculus as the basis for a practical tool. The main disadvantage of formula-based variation is that many operations require solving satisfiability problems to complete. However, multiple researchers have observed that this does not seem to be a problem in practice since SAT solvers are fast at checking the kinds of formulas that arise in variational software [17, 20].

7. Conclusions and Future Work

This paper presented the formula choice calculus, a generalization of the choice calculus that replaces atomic dimensions as conditions on choices with propositional formulas of dimensions. This more expressive notation supports more efficient (i.e. containing less redundancy) variation representations without introducing complicated naming constructs. It also provides a closer mapping to existing variation languages used in practice.

The main advantage of using the choice calculus is the ability to reuse its associated theory. The most important and useful part of this theory is the equivalence relation on choice calculus terms. Although FCC has appeared and been used in previous work, the equivalence rules had only been fully defined and proven sound in terms of ACC. This work fills this gap, defining many equivalence rules for FCC and proving their soundness. This validates previous work that has already used FCC, and also suggests new semantics-preserving transformations on formula-based representations going forward.

In future work, we plan to identify minimal complete subsets of the equivalence rules. A complete set of rules is one that ensures that any two semantically equivalent expressions are syntactically related using only rules in the set. There is existing work on identifying complete sets of axioms for Boolean logic [22] that can be directly reused here. Obviously, complete sets of equivalence rules are not unique since many of the rules are mutually derivable.

Acknowledgments

This work is supported by AFRL Contract No. FA8750-16-C-0044 (Raytheon BBN Technologies) under the DARPA BRASS program.

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, New York, 2004.

[2] S. Chen and M. Erwig. Guided Type Debugging. In *Int. Symp. on Functional and Logic Programming*, LNCS 8475, pages 35–51, 2014.

[3] S. Chen and M. Erwig. Early Detection of Type Errors in C++ Templates. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 133–144, 2014.

[4] S. Chen and M. Erwig. Principal Type Inference for GADTs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 416–428, 2016.

[5] 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.

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

[7] S. Chen, M. Erwig, and E. Walkingshaw. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming*, pages 6:1–6:28, 2016.

[8] D. S. Dummit and R. M. Foote. *Abstract Algebra*. John Wiley and Sons, third edition, 2004.

[9] 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, 2011.

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

[11] M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering*, pages 55–99, 2012.

[12] 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, 2012.

[13] 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, 2011.

[14] 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, 2012.

[15] 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.

[16] 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.

[17] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering*, pages 81–91, 2013.

[18] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *IEEE Int. Conf. on Software Engineering*, pages 380–391, 2015.

[19] J. Meinicke. VarexJ: A Variability-Aware Interpreter for Java Applications. Master's thesis, University of Magdeburg, 2014.

[20] M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Int. Software Product Line Conf.*, pages 231–240, 2009.

[21] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *IEEE Int. Conf. on Software Engineering*, pages 907–918, 2014.

[22] T. Ninomiya and M. Mukaidono. Complete and Independent Sets of Axioms of Boolean Algebra. In *Int. Symp. on Multiple-Valued Logic*, pages 169–174, 2003.

[23] Ş. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *IEEE Int. Conf. on Software Maintenance and Evolution*, 2016. To appear.

[24] E. Walkingshaw. *The Choice Calculus: A Formal Language of Variation*. PhD thesis, Oregon State University, 2013. `http://hdl.handle.net/1957/40652`.

[25] E. Walkingshaw and K. Ostermann. Projectional Editing of Variational Software. In *ACM SIGPLAN Int. Conf. on Generative Programming and Component Engineering*, pages 29–38, 2014.

[26] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational Data Structures: Exploring Trade-Offs in Computing with Variability. In *ACM SIGPLAN Symp. on New Ideas in Programming and Reflections on Software (Onward!)*, pages 213–226, 2014.