A Calculus for Modeling and Implementing Variation

Eric Walkingshaw
School of EECS
Oregon State University
walkiner@eecs.oregonstate.edu

Martin Erwig
School of EECS
Oregon State University
erwig@eecs.oregonstate.edu

ABSTRACT

We present a formal calculus for modeling and implementing variation in software. It unifies the compositional and annotative approaches to feature implementation and supports the development of abstractions that can be used to directly relate feature models to their implementation. Since the compositional and annotative approaches are complementary, the calculus enables implementers to use the best combination of tools for the job and focus on inherent feature interactions, rather than those introduced by biases in the representation. The calculus also supports the abstraction of recurring variational patterns and provides a metaprogramming platform for organizing variation in artifacts.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software; D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages, specialized application languages*

General Terms

Design, Languages, Theory

Keywords

choice calculus, feature-oriented software development, preprocessors, separation of concerns, variational software

1. INTRODUCTION

In general, there are two ways to encode variability in software. Variation can be captured in-place by *annotating* the parts of the software that differ, or variable parts can be separated and later *composed* into a working system [11]. The complementary nature of these approaches, explored in Section 2, is evident when considering how to represent overlapping variability in multiple dimensions, called *feature interactions* [19]. Compositional approaches excel when interactions are widespread and regular, while annotative representations are best suited for a small number of irregular interactions. Kästner, Apel, and their collaborators have explored these trade-offs in depth [9–13] and identified the need for a way "to combine annotation-based and composition-based approaches in a unified and efficient framework" [10].

In this paper, we present a calculus that generalizes and unifies the compositional and annotative approaches to representing variation. Section 3 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. The calculus is more powerful than simply adding annotative variation to compositional components, however. In Section 4, 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 main contributions of this paper are: (1) The *compositional choice calculus* (CCC), a formal language for representing, generating, and organizing variation in tree-structured artifacts that achieves the goals stated above. The language is developed in Sections 3 and 4, and its syntax is given in Section 5. (2) A formal semantics for CCC, given in Section 6. The semantics is interesting because it ensures the hygiene property [15] of variation abstractions through a novel compositional semantics definition, rather than by the traditional renaming strategy. (3) A theoretical analysis of the local expressiveness [7] of CCC relative to compositional and annotative representations, given in Section 7. This demonstrates that CCC is more locally expressive than either approach alone, and than a simple union of the two. Throughout the paper 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. Our long-term goals concern the study of variation in general and the development of languages and theories that can be reused in a range of areas where variability arises. After a discussion of related work in Section 8, Section 9 discusses these broader goals.

2. BACKGROUND

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 creat-

This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grant CCF-0917092.

```
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 1: A small integer buffer SPL.

```
refines 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 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 2: Class refinements implementing undo features.

```
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 3: Buffer with annotated logging and undo features.

ing 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 [1].

Variability is expressed in FOSD at two distinct levels. *Feature modeling* describes the high-level relationships between conceptual features in the problem domain [8]. *Feature implementation* associates conceptual features with the code and other artifacts that realize them in the solution domain. There are broadly two competing approaches to feature implementation: compositional and annotative. These will be described in the next two subsections.

2.1 Compositional Approaches

Compositional approaches are strongly motivated by traditional software engineering pursuits like separation of concerns and stepwise refinement [2, 18, 19]. 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 [2, 3] or aspects [4, 18]. 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 1 shows a simple SPL in the compositional style. This running example is based on an example from Liu et al.'s 2006 ICSE paper [17]. The base program $\underline{b}$[1] is a simple integer buffer written in Java. We add to this an optional logging feature $\underline{l}$, implemented as an aspect in the AspectJ language [14]. 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 $\underline{b}$ and the buffer with the logging feature added, obtained by applying $\underline{l}$ to $\underline{b}$, which we write $\underline{l} \bullet \underline{b}$.

In Figure 2, we implement two possible undo features as *class refinements* in the Jak language of the AHEAD Tool Suite [2].[2] 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. 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 the $\bullet$ operator is overloaded—its implementation depends on the types of its arguments. In $\underline{l} \bullet \underline{b}$, the operator represents aspect weaving [4]; in $\underline{u_o} \bullet \underline{b}$ it represents class refinement. This makes it possible to extend the compositional approach to new languages and artifact types by simply adding new instances of the $\bullet$ operator [2].

2.2 Annotative Approaches and the Choice Calculus

Annotative approaches represent variation in-place, directly marking code to be conditionally included if the corresponding features are selected. In our previous work we have developed a formal language for representing annotative variation called the *choice calculus* [5]. We call the language introduced in this paper the *compositional* choice calculus because it adds compositional features to this annotative core. Likewise, when the distinction is significant, we call the original calculus the *annotative* choice calculus.

Figure 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 previous subsection—it only logs changes to the buffer's value, and prints a unique message for each method.

In the choice calculus a variation point is captured by a *choice* $D\langle e_1, \ldots, e_n \rangle$ between n alternative expressions, associated with a

[1]Underlined names indicate *plain* expressions, as defined in Section 5. This distinction can be safely ignored for now.

[2]We omit the `class` keyword from Jak refinements to save space.

dimension D. Dimensions synchronize the selection of alternatives from different choices. A dimension declaration **dim** $D\langle t_1,\ldots,t_n\rangle$, declares a new dimension D with *n tags*. If tag t_i is selected from D, then every choice bound by D will be replaced by its ith alternative e_i, and the dimension declaration will be removed. Our example contains two dimensions, *Log* and *Undo*. All choices bound by 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 the two possible undo features and the case where no undo feature is included. To get an integer buffer with no logging and one-step undo, we can select *Log.no* and *Undo.one*.

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 might formally represent the buffer's `get` method as `get`$\prec$`()`,`return`$\prec$`buff`$\succ\succ$. We rarely show this tree structure explicitly, however, preferring to embed the notation directly in the concrete syntax of the program, for readability.

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 `1+`$D\langle$`2`$,\circ\rangle$ is invalid since $\circ$ is not syntactically correct at the position of the choice in the surrounding Java expression.

The choice calculus is not intended to be used directly in software but rather as a formal basis for theoretical research on software variation. In our previous work, we define a set of semantics-preserving laws for transforming choice calculus expressions, identify normal forms and show that they can be reached, and develop quality criteria for variational artifacts [5]. Many of these results can be reused directly in the compositional choice calculus presented in this paper.

2.3 Representing Feature Interactions

The salient problem in FOSD is detecting, resolving, and managing the interactions of a huge number of conditionally included features [13]. This is a large problem that spans all stages of the software life cycle. Here we consider only the smaller subproblem of *representing* intended feature interactions in a way that is structured and manageable.

Interactions are represented quite differently in the two approaches to feature implementation. In the annotative approach exemplified by the choice calculus, interactions appear as nested choices. For example, the *Log* choices inside the *Undo* choice in Figure 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. 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 representation of the logging feature $\underline{l}$ as an aspect in Figure 1(b) demonstrates how regular interactions can be modularized in the compositional approach. 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.

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

(a) Add logging to $\underline{u_o}$, $\underline{l_{u_o}}$.

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

(b) Add logging to $\underline{u_m}$, $\underline{l_{u_m}}$.

Figure 4: Modularized feature interactions.

The logging feature implemented in Figure 3 is less regular, however, and 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.

A solution to the problem is described by Liu et al. [17] and demonstrated in Figure 4. Essentially, we split the representation of the logging feature into several smaller refinements. The refinement $\underline{l_b}$ (not shown) 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 [17]

2.4 Toward the Compositional Choice Calculus

The next two sections describe the development of CCC in two steps. Section 3 integrates the compositional and annotative approaches to feature implementation by applying the choice calculus to compositional components and to the feature algebras used to assemble these components. Section 4 introduces variation abstractions to encapsulate variation patterns and to extend the choice calculus with metaprogramming capabilities. The examples in these two subsections demonstrate how CCC addresses existing problems in FOSD.

Combining these extensions with the basic choice calculus from Section 2.2 produces CCC, whose syntax is given in Section 5 and semantics in Section 6.

3. INTEGRATING THE TWO APPROACHES

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 [9–13]. These trade-offs are evident in our integer buffer SPL. The separated undo features $\underline{u_o}$ and $\underline{u_m}$ in Figure 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

```
dim Undo⟨one,many⟩ in
refines 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 5: Undo refinement u with annotative variation.

in Figure 3 requires direct modification of the base program and clutters its definition with code that is only sometimes relevant.[3] However, the compositional undo features contain redundancy not found in the annotative representation, complicating their maintenance. 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 5 presents an obvious compromise, where we annotate compositional feature implementations. This allows the two versions of undo 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 $\underline{u_m}$, and introducing a new dimension *Undo* to capture their differences in synchronized choices. In fact, the choice calculus makes it possible to derive u from $\underline{u_o}$ and $\underline{u_m}$ mechanically. By applying the transformation laws [5], we can automatically transform the choice calculus expression $\mathbf{dim}\ Undo\langle one, many\rangle\ \mathbf{in}\ Undo\langle \underline{u_o}, \underline{u_m}\rangle$ into u. We say that these two expressions are *equivalent* ($\equiv$).

Annotations are also useful 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, we will not log calls to methods in the undo feature since these are added only after the logging aspect is woven in. This makes assembling components into products potentially error-prone.

We can write a single expression that adheres to these constraints and describes all of the products that can be generated using annotations. 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.

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

Note that the u component itself contains annotational variation in a dimension named *Undo*. Each dimension declaration defines a new statically scoped dimension of variation, so these two *Undo* dimensions are different and can be selected independently. 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 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.

[3]Better user interfaces can alleviate some of the readability concerns [10, 16].

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 two refinements $\underline{l_{u_o}}$ and $\underline{l_{u_m}}$ in Figure 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 alternately 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 can be left up to the features' implementors.

4. ADDING VARIATION ABSTRACTIONS

In this section we further the development of CCC by introducing an abstraction construct and by generalizing feature composition to function application. These changes support reuse, the minimization of redundancy, and extend the choice calculus with metaprogramming capabilities. This section motivates these extensions through examples.

4.1 Reusable Optional Wrappers

A remaining bit of redundancy in our integer buffer example is the repetition of the `undo` method declaration in two alternatives of the *Undo* choice in Figure 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 the annotative choice calculus we provide a static sharing construct of the form $\mathbf{share}\ v = e'\ \mathbf{in}\ e$.[4] Using this, we can factor the redundancy out as follows, where b_o and b_m refer to the body of the `undo` method corresponding to the undo-one and undo-many features, respectively.

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

This solution is troublingly inelegant. The problem is related to the *optional wrapper* problem encountered by Kästner et al. in the development of their CIDE tool [13]. An optional wrapper is a variation pattern where the goal is to conditionally wrap an expression in another construct, for example, a method declaration, 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 with $\mathbf{share}\ v = e\ \mathbf{in}\ D\langle \texttt{try\{}v\texttt{\}catch\{}\ldots\texttt{\}}, v\rangle$—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

[4]We call this construct **let** in [5] but name it differently here to prevent confusion since it behaves differently than traditional **let**-expressions.

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 $\lambda b.$`void undo(){`b`}`. We can then rewrite the optional `undo` method from Figure 3 more simply as $\mathit{Undo}\langle u_w\ b_o, u_w\ b_m, \circ\rangle$.

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). Rather than just factoring redundancy, this makes it 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.

4.2 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.

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

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 alternation 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 and 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*.

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

The function accepts a feature f and a program b, and recursively applies f to b an arbitrary number of times. Each time the *yes* tag is selected from *Arb*, a new copy of the 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.

5. THE COMPOSITIONAL CHOICE CALCULUS

The syntax of the compositional choice calculus is given in Figure 6. The first three constructs are from the annotative choice calculus [5] described in Section 2.2. The first encodes the tree-structure of the object language, choices introduce variation points within that structure, and dimensions scope and synchronize choices and organize the variation space. The next three constructs replace the static sharing constructs of the choice calculus with the separable, dynamic metaprogramming constructs introduced in the previous section. These are as in the lambda calculus.

In the following discussion, it will often be useful to talk about expressions that do not include a particular syntactic category s—

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

Figure 6: Syntax of the compositional choice calculus.

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, abstractions, or any other syntactic category). Further, we say that an expression is *variation free* if it is dimension free and choice free; a *sharing free* expression is abstraction free, application free, and reference free. Finally, an expression is called *plain* if it is variation free and sharing free. A plain expression therefore consists only of structure nodes, representing a plain artifact in the object language. We indicate that an expression $\underline{e}$ is plain by underlining it. This notation is used for the examples in Sections 2 and 3.

Note that we do not syntactically restrict the left-hand side (LHS) of applications to function 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. Application can be viewed as a generalization of the overloaded feature composition operator $\bullet$ from Section 2.1. As a special case, when we apply two plain expressions $\underline{e}\ \underline{e}'$, we defer to the instance of $\bullet$ determined by $\underline{e}$ 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, 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 an extension of the semantics-preserving transformation laws for choice calculus [5]. For distributing across dimensions and choices in the LHS of an application, we have the following laws.

$$\textsc{App-Dim-L}\quad \frac{D \notin FD(e')}{(\mathbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \mathbf{in}\ e)\ e' \equiv \mathbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \mathbf{in}\ e\ e'}$$

$$\textsc{App-Chc-L}\quad (D\langle e_1,\ldots,e_n\rangle)\ e' \equiv D\langle e_1\ e',\ldots,e_n\ e'\rangle$$

The function $FD(e)$ returns the set of *free dimensions* in e, that is, the dimensions of unbound choices. Since we change the scope of the dimension D, the premise in App-Dim-L prevents the capture of choices in e'.

A symmetric law App-Chc-R exists for distributing across choices in the RHS of an application. However, there is no App-Dim-R since dimension declarations in the RHS of an application can be duplicated during β-reduction, producing conceptually distinct dimensions.

There is also a straightforward law for commuting abstraction and choice constructs.

$$\textsc{Abs-Chc}\quad \lambda v.D\langle e_1,\ldots,e_n\rangle \equiv D\langle \lambda v.e_1,\ldots,\lambda v.e_n\rangle$$

There is not, however, a law for commuting abstractions and dimensions. The *arb* example in Section 4.2 demonstrates how a dimension declaration within an abstraction can be applied multiple times to produce different dimensions.

To demonstrate the evaluation of an expression, consider an integer buffer SPL that optionally applies the undo-one feature $\underline{u_o}$ to the buffer $\underline{b}$, then applies the logging feature $\underline{l}$: $\underline{l}\ (opt\ \underline{u_o}\ \underline{b})$. If we expand *opt* and perform β-reduction twice to consume its arguments, we get the following expression in which the annotations are more obvious.

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

Selecting *Opt.yes* yields a buffer with both the undo-one and logging features included: $\underline{l}\ (\underline{u_o}\ \underline{b}) \equiv \underline{l} \bullet \underline{u_o} \bullet \underline{b}$. Selecting *Opt.no* yields a 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. Intuitively, the meaning of a compositional choice calculus expression is the total set of variants it represents and the decisions that lead to those variants. We formalize the relationship between decisions and the variants they produce in the next section.

6. FORMAL 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. We define the semantics of a CCC expression to be a mapping from decisions to the plain artifacts they eventually produce.

Computing this mapping is complicated by the fact that computations can duplicate and remove dimension declarations, so we cannot statically determine the decisions that must be made by simply looking at a 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 the user, then proceed reducing based on their response. For the purpose of defining a formal, 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 three subsections, we consider these components of evaluation separately. Section 6.1 describes the process of dimension elimination. This is based on our previous work on the annotative choice calculus [5]. The semantics of reduction is defined compositionally. Given an expression $e\ e'$, we compute the *partial semantics* of e, the partial semantics of e', then compose the results. Section 6.2 defines the structure of a partial semantics mapping and how to compose them; Section 6.3 defines how to compute the partial semantics of an expression. The semantics is defined in this way for two reasons: (1) to ensure that we only ever invoke the $\bullet$ operator on plain expressions, and (2) to avoid the problem of *choice capture*, which is similar to the hygiene problem of traditional macro systems [15].

6.1 Dimension and Choice Elimination

Formally, a *decision* is a sequence of *qualified tags*, where a qualified tag $D.t$ is a tag t prefixed by its dimension D. We use q to range over qualified tags, $\bar{q}$ to indicate a sequence of qualified tags (that is, a decision), and ε to represent the empty decision containing no tags. Finally, we use concatenation $q\bar{q}$ to prepend a tag q to an existing decision $\bar{q}$, and to concatenate two decisions $\bar{q}$ and $\bar{q}'$, as $\bar{q}\bar{q}'$.

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. For example, in the semantics of the following expression (listed explicitly as a set of decision/plain-expression pairs) tags in dimension A always appear before tags in dimension B since the declaration of A occurs before the declaration of B.

$$\begin{aligned}\llbracket \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ \mathbf{dim}\ B\langle c,d\rangle\ \mathbf{in}\ A\langle B\langle 1,2\rangle, B\langle 3,4\rangle\rangle\rrbracket = \\ \{((A.a,B.c),1),((A.a,B.d),2), \\ ((A.b,B.c),3),((A.b,B.d),4)\}\end{aligned}$$

This ordering constraint is needed since the meaning of an expression can change if a tag is selected prematurely. For example, function application can multiply a single declaration into many independent dimensions, as demonstrated by the *arb* example in Section 4.2, but if the dimension is eliminated before the application is reduced, the dimensions will become effectively synchronized.

Strictly ordering tag selection also reduces unnecessary selections and redundant entries in the semantics. Consider the following, in which dimension B is sometimes eliminated by an upstream selection in dimension A.

$$\begin{aligned}\llbracket \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 1, \mathbf{dim}\ B\langle c,d\rangle\ \mathbf{in}\ B\langle 2,3\rangle\rangle\rrbracket = \\ \{(A.a,1),((A.b,B.c),2),((A.b,B.d),3)\}\end{aligned}$$

When selecting tag $A.a$, the B dimension is eliminated, and so does not appear in the decision. In the other cases, when $A.b$ is chosen, the B dimension remains, so a tag is also selected from B to produce the final variants. In this situation, we say that dimension B is *dependent* on the selection of $A.a$.

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. When computing the semantics, each of these steps but the third is handled by the partial semantics function defined in Section 6.3. We call the third step *choice elimination* and define it as follows. Given a dimension declaration $\mathbf{dim}\ D\langle t_1,\ldots,t_n\rangle$ 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 7(a). Most cases just propagate the selection to their subexpressions. There are two interesting cases: (1) Recursion ceases if another dimension declaration named D is encountered, preserving local dimension scoping. (2) 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 of the same name, as in $D\langle D\langle 1,2\rangle,3\rangle$, and they will both be eliminated when we make a selection in D. This makes it impossible 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 previous work, we provide strategies for removing dead alternatives and other kinds of dead subexpressions [5].

6.2 Composing Partial Semantics

In Section 5 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 an unbound choice: $(\lambda x.\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ x)\ A\langle 1,2\rangle$. Applying β-reduction brings the choice within the scope of the dimension declaration: $\mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 1,2\rangle$. We call this phenomenon *choice capture*, and it is highly undesirable since it breaks the static scoping of dimension names. The situation is analogous to the hygiene issue in other metaprogramming systems [15].

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

(a) Choice elimination (Section 6.1).

$$
\begin{aligned}
V_\rho(v) &= \rho(v) \\
V_\rho(e\ e') &= V_\rho(e) \bowtie V_\rho(e') \\
V_\rho(\lambda v.e) &= \{(\varepsilon, (\lambda v.e, \rho))\} \\
V_\rho(a\prec\succ) &= \{(\varepsilon, a\prec\succ)\} \\
V_\rho(a\prec e^n\succ) &= \{(\bar{q}^n, a\prec e'^n\succ) \mid ((\bar{q}_i, e'_i) \in V_\rho(e_i))^{i:1..n}\} \\
V_\rho(\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e) &= \{(D.t_i\bar{q}, e') \mid i \in \{1,\ldots,n\}, \\
&\qquad (\bar{q}, e') \in V_\rho(\lfloor e\rfloor_{D.i})\}
\end{aligned}
$$

(b) Computing partial semantics (Section 6.3).

Figure 7: Definitions used in computing the semantics of a compositional choice calculus expression.

In order to avoid the problem of choice capture, we do not perform β-reduction directly. Instead, when computing the semantics of an application, we compute the partial semantics (defined below) of the left and right expressions separately, then compose the results. For our purposes, this is better than the standard renaming solution to the hygiene problem, since we want to preserve the given names of dimensions, which appear in the decisions contained in the semantics.

This strategy is also crucial for the reuse of existing compositional feature implementation tools (such as AHEAD) in a mixed annotative/compositional setting. Given a feature f and a base program b, we can evaluate $f\ b$ even if f and b both contain annotations. Essentially, we will compute the partial semantics of f and b separately, and then invoke the $\bullet$ operator on every pairwise combination of results. This ensures that we only ever invoke $\bullet$ on plain expressions, allowing us to mix annotations into our compositional components, without modifying the existing tools.

The partial semantics, S, of an expression is a mapping from decisions to *values*. A value φ is either a plain expression, or a *closure*, $\varphi ::= \underline{e} \mid (\lambda v.e, \rho)$. 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 values. We use the notation $(v,S) \oplus \rho$ to represent mapping variable v to the partial semantics S in the environment ρ. Finally, 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, with partial semantics $V_\rho(e_l) = S_l$ and $V_\rho(e_r) = S_r$, we can compute the application of e_l to e_r by composing S_l and S_r. We write this as $S_l \bowtie S_r$. In general, e_l and e_r can be arbitrarily large choice calculus expressions, representing potentially many variants each. Conceptually, composition corresponds to a pairwise application of every variant from the partial semantics of e_l to every variant from the partial semantics of e_r. Formally, composition proceeds by iterating over entries in S_l and calling a helper function, $\lhd$, as below.

$$S_l \bowtie S_r = \{(\bar{q}_l, \varphi_l) \lhd S_r \mid (\bar{q}_l, \varphi_l) \in S_l\}$$

Each entry from S_l consists of a decision $\bar{q}_l$ and a value φ_l. There are two cases to consider: either φ_l is a plain expression or φ_l is a closure. If φ_l is a plain expression, then every value $\varphi_r \in rng(S_r)$ must also be a plain expression, and we compose φ_l with each using the object composition operator $\bullet$. We also concatenate the decision that produced φ_l with the decision that produce φ_r, to create the decision which produces the combined expression.

$$(\bar{q}_l, \underline{e}'_l) \lhd S_r = \{(\bar{q}_l\bar{q}_r, \underline{e}'_l \bullet \underline{e}'_r) \mid (\bar{q}_r, \underline{e}'_r) \in S_r\}$$

Note that if φ_l is a plain expression but there is a value φ_r that is *not* a plain expression, then the semantics is undefined.

Considering the second case, if φ_l is a closure $(\lambda v.e, \rho)$, then 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 and add each to our composed partial semantics.

$$(\bar{q}_l, (\lambda v.e, \rho)) \lhd S_r = \{(\bar{q}_l\bar{q}', e') \mid (\bar{q}', e') \in V_{(v,S_r)\oplus\rho}(e)\}$$

Without the definition of $V_\rho(e)$, it is hard to verify that this does what we expect, but we expect each occurrence of v in e to be able to take on any possible variant in S_r.

6.3 Computing the Semantics

The final piece needed to define the semantics of CCC expressions is the function $V_\rho(e)$, which computes the partial semantics of e in the context of the environment ρ. The definition of V_ρ is given in Figure 7(b). Because of the groundwork laid in the previous subsections, there should be few surprises.

For the three lambda calculus constructs, V_ρ is very straightforward: for references, it performs an environment look up; for applications, it computes the partial semantics of each subexpression and composes the results; and for abstractions, it produces a trivial mapping to a closure. If an unbound variable is encountered, lookup will fail and the semantics is undefined.

The cases for structures are similarly straightforward despite the dense notation. 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. The notation $(x_i)^{i:1..n}$ can be expanded to $x_1, \ldots, x_n$, and the notation x^n implies the concatenation of every x_i.

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 eliminates all bound choices. If a choice is unbound, the entire semantics is undefined.

Finally, we can use V_ρ 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.

7. RELATIVE LOCAL EXPRESSIVENESS

We have claimed that the compositional choice calculus subsumes the annotative and compositional approaches to feature implementation and that it is indeed more powerful than either approach on its own. We have provided example-based evidence of these claims throughout the paper. In this section, we make these comparisons more formally and directly, using Felleisen's framework for comparing the relative local expressiveness of languages [7]. 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 have the same semantics according to the semantics definitions of their respective languages [7].

We compare three languages: the compositional choice calculus (CCC), the annotative choice calculus (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 [2] (see Section 2.1), 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 1. CCC *is more locally expressive than* CFA.

PROOF OF LEMMA 1. 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 variation-free (see Section 5).

$$e_{\text{CCC}} = \textbf{dim}\ D\langle t_1, t_2\rangle\ \textbf{in}\ D\langle e_1, e_2\rangle$$

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 2. CCC *is more locally expressive than* ACC.

PROOF SKETCH OF LEMMA 2. 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.

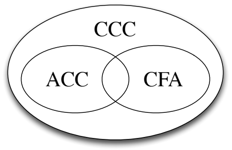
To show that there is no loss of expressiveness from ACC to CCC, we must provide a local transformation from e_{ACC} to an operationally equivalent CCC expression. We only describe this transformation at a high level here. The individual steps, however, are just applications of the semantics-preserving transformation laws for ACC expressions, defined and proved correct in our previous work [5]. (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, then CCC can *macro express* ACC [7]. 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 3. CCC *is more locally expressive than* ACC ∪ CFA.

PROOF OF LEMMA 3. This follows directly from Lemma 1 and Lemma 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 ∩ 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 at right, which illustrates the relative local expressiveness 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.

8. RELATED WORK

Section 2 provides an overview of FOSD, the different approaches to feature implementation, and the trade-offs they present for representing feature interactions. In this section, we focus on other attempts at combining the benefits of compositional and annotative variation representations.

Kästner and Apel have suggested that their annotative CIDE tool [11] could be integrated with the compositional AHEAD tool suite [2], and discuss the implications of such a merger [9]. Many of these implications we have not considered here; for example, that an integrated model can support migrating from one implementation approach to another. Elsewhere, they propose the idea of a "virtual separation of concerns", which attempts to bring the maintenance and understandability benefits of separability to annotative approaches through tool support for working with projections of annotated artifacts [10]. With Kuhlemann, they have created $\mathrm{LJ}^{\mathrm{AR}}$, a formal language for combining annotative and compositional variation in Lightweight Java programs [12]. While our transformation laws describe the commutation of annotations *with* and *within* generic compositional components, $\mathrm{LJ}^{\mathrm{AR}}$ supports refactorings for moving *between* the two implementation approaches, but in a way that is necessarily tied to a specific object language.

The *XML Variant Configuration Language* (XVCL) [20] is another language-based attempt at merging the annotative and compositional approaches. Like CPP (but unlike the choice calculus and CIDE), its in-place variation annotations are structurally undisciplined. Distributed variation is supported through named "breakpoint" annotations, where code specified elsewhere can be automatically inserted. While this provides separability, the need to insert breakpoint annotations means that XVCL does not support stepwise refinement, a core tenet of compositional approaches. (Though the sometimes necessary "hook" method technique [17] violates this in purely compositional approaches as well.) This makes separability in XVCL more similar to **share**-expressions in the choice calculus than to compositional approaches.

9. CONCLUSIONS

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. The variational fixed point combinator *arb*, from Section 4.2, suggests a new model of interactive variational computation, where computations dynamically produce decision points that will affect the subsequent computation.

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* [6], which is concerned with writing programs to generate, query, manipulate, and analyze variation structures.

10. REFERENCES

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.

[3] 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.

[4] T. Elrad, R. E. Filman, and A. Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44(10):28–32, 2001.

[5] 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.

[6] M. Erwig and E. Walkingshaw. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering*, 2011. To appear.

[7] M. Felleisen. On the Expressive Power of Programming Languages. *Science of Computer Programming*, 17(1–3):35–75, 1991.

[8] 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.

[9] 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.

[10] C. Kästner and S. Apel. Virtual Separation of Concerns—A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[11] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*, pages 311–320, 2008.

[12] 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, 2009.

[13] 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, 2009.

[14] 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.

[15] E. Kohlbecker, D. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion. In *ACM Conf. on LISP and Functional Programming*, pages 151–161, 1986.

[16] 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.

[17] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *IEEE Int. Conf. on Software Engineering*, pages 112–121, 2006.

[18] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[19] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conf. on Object-Oriented Programming*, pages 419–443, 1997.

[20] H. Zhang and S. Jarzabek. XVCL: A Mechanism for Handling Variants in Software Product Lines. *Science of Computer Programming*, 53(3):381–407, 2004.