Adding Configuration to the Choice Calculus

Martin Erwig
Oregon State University

Klaus Ostermann
University of Marburg

Tillmann Rendel
University of Marburg

Eric Walkingshaw
Oregon State University

ABSTRACT

The choice calculus, a formal language for representing variation in software artifacts, features syntactic forms to map dimensions of variability to local choices between source code variants. However, the process of selecting alternatives from dimensions was relegated to an external operation. The lack of a syntactic form for selection precludes many interesting variation and reuse patterns, such as nested product lines, and theoretical results, such as a syntactic description of the configuration process.

In this paper we add a selection operation to the choice calculus and illustrate how that increases the expressiveness of the calculus. We investigate some alternative semantics of this operation and study their impact and utility. Specifically, we will examine selection in the context of static and dynamically scoped dimension declarations, as a well as a linear and comprehensive form of dimension elimination. We also present a design for a type system to ensure configuration safety and modularity of nested product lines.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software Configuration Management*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

Keywords

Choice calculus, semantics, modularity

1. INTRODUCTION

The need to maintain variation in software artifacts is widely acknowledged, and has led to the development of many models, methods, and tools. The topic has been approached from different angles, and a diverse set of perspectives and subfields have emerged as a result (for example, software product lines, feature-oriented software development, and configuration management). While there are several formal approaches in some of these subfields, a widely accepted, general formal model of variation that applies across a wide range of fields does not exist so far.

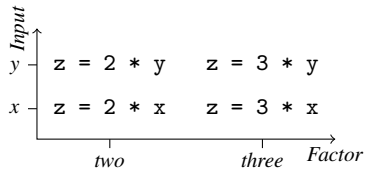
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VAMOS January 23–25, 2013, Pisa, Italy
Copyright 2013 ACM 978-1-4503-1541-8/13/01 ...$15.00.

One such general model that has been proposed is the choice calculus [6]. It provides a formal foundation for the study of *some aspects* of variability in software [5]. As a calculus, it can serve as a vehicle for the exploration of the meta-theory of variation, as a backend (or core language) for the implementation and comparision of more practical tools, and as an implementation language for smaller case studies. We expect that a layer of syntactic sugar on top of the choice calculus would enable its use also for larger case studies and industrial projects, but in the present paper, we focus on the core calculus. Specifically, the choice calculus addresses three facets of variation: (1) the organization of the variation space into dimensions of variability, (2) the introduction of choices to capture points of variation, and (3) a selection process to generate a particular variant. The prior work has focused on dimensions and choices as first-class elements of the choice calculus and relegated selection to a second-class operation that is only available externally, as a meta-theoretic operation. This asymmetry is apparent in the syntax of the choice calculus, which provides syntactic forms for declaration and choice, but not for selection.

The lack of a syntactic form for selection is unfortunate from both practical and theoretical perspectives. From a practical perspective, we cannot model software systems that fully or partly configure themselves, such as build scripts for variational software systems, nested product lines, computed configurations, or wrappers that rename dimensions of variation. From a theoretical perspective, we cannot explore the meaning of selection with syntactic means such as rewriting rules or type systems.

To expand the scope of the choice calculus as a vehicle to study variation, we propose to extend it with a selection operation. Specifically, this paper makes the following contributions.

- We review the choice calculus with a focus on its formal structure in Section 2 where we also explore the structural similarities to the λ-calculus.
- We extend the choice calculus with a syntactic form for selection. Section 3 introduces the syntax and an intuition for the semantics and illustrates how selection supports renaming and modularity.
- To better understand the requirements for a selection semantics we discuss in Section 4 a series of design questions that explore how a selection behaves in particular situations and how it interacts with other choice calculus constructs.
- We consider the design of a type system for the extended choice calculus. Section 5 introduces a notion of configuration safety and explores the differences between a flat and a nested notion of configuration.
- In Section 6 we explore how a variation of the type system can be employed to guarantee a form of information hiding for nested product lines.



(a) Variants.

```
dim Factor⟨two, three⟩ in
dim Input⟨x, y⟩ in
    z = Factor⟨2,3⟩ * Input⟨x,y⟩
```

(b) Choice-calculus expression.

```
z = dim Factor⟨two, three⟩ in
    dim Input⟨x, y⟩ in
        Factor⟨2,3⟩ * Input⟨x,y⟩
```

(c) Alternative representation.

Figure 1: Modeling dimensions of variation.

2. THE CHOICE CALCULUS

The choice calculus [5, 6] is a formal language for variational software artifacts that organizes the configuration space of a variational software system into *dimensions*. Each dimension models one of the decisions that must be made to obtain a particular variant. For example, Figure 1(a) shows four variants of an assignment software snippet that varies in two independent dimensions: One dimension is the name of the variable to read, and the other dimension is the factor to multiply with. These two independent dimensions of variation give rise to four different variants. Since the two dimensions are independent of each other, the shown variation illustrates the challenge of "combinatorial explosion" that is typical for variability: A relatively small number of configuration options induces a potentially huge number of variants. If we want to store and analyze variational software artifacts efficiently, we better avoid such enumerations of all variants.

2.1 Modeling Variation by Choices

The choice calculus helps to avoid combinatorial explosion by allowing local *choices* between the fragments of a software artifact that actually vary. For example, the code in Figure 1(b) represents all four variants discussed above in a single choice-calculus expression. The common parts "`z =`" and "`*`" are stated only once, and the dimensions are disentangled because the associated choices are as local as possible.

A choice expression such as $Factor\langle 2,3\rangle$ means that depending on the configuration for dimension *Factor*, we choose between the code fragments `2` and `3`. In a well-formed choice-calculus expression, all dimension names used in choice expressions have to be bound by explicit *dimension declarations*, and the number of alternative code snippets in the choices have to match the number of tags declared for that dimension. For example, the expression in Figure 1(b) is well-formed, because both dimensions used in the choice expressions are bound in the surrounding dimension declarations. The declaration **dim** *Factor*⟨*two*, *three*⟩ **in** introduces the *Factor* dimension with its tags *two* and *three*. The dimension name is in scope for choice expressions that follow the **in** keyword. The association of alternatives in choice expressions to tags in dimension declarations is by position: the first alternative `2` belongs to the first tag *two*, and so on.

Dimension declarations can also occur nested inside the expression. For example, the expressions in Figures 1(b) and (c) mean the same. The choice calculus allows us to state and proof such equivalences between expressions formally. In prior work we have proposed more equivalences as well as normalization procedures [6]. To formally verify an equivalence, we have to also define a semantics for the choice calculus.

2.2 Modeling Configuration by Tag Selection

The semantics of the choice calculus is based on the notion of configuring an expression by *selection* of tags for each dimension used in the expression. This incremental process resolves a choice-calculus expression into a concrete program variant. Selection of a qualified tag $D.t$ proceeds in two steps: (1) The first (that is, topmost, leftmost) dimension declaration that introduces D is located in the expression. (2) All choices bound by that dimension declaration are replaced by the alternative that corresponds to the position of the selected tag t in the declaration for D. For example, Figure 2 shows some example selections for our running example. Following the left path, after selection of *Input.x* and *Factor.three*, we have fully configured the variational software artifact. We can also reach the same variant with the opposite order of selections. Both sequences of selections would resolve the alternative expression in Figure 1(c) to the same variant. This observation is the basis of the formal notion of equivalence between choice calculus expressions.

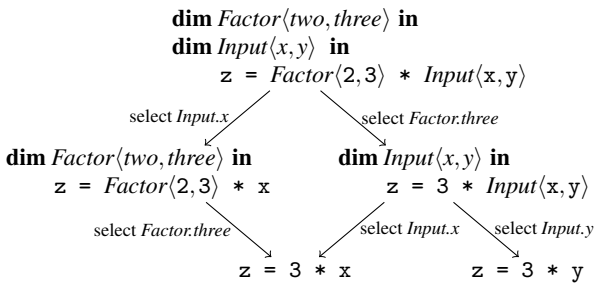


Figure 2: Selecting a variant.

Formally, the semantics of each choice calculus expression is defined as a mapping from sequences of tags to plain expressions, and the semantic function $[\![e]\!]$ produces this mapping [6]. Tag selection can thus be expressed as function application. Therefore, if e represents the choice calculus expression shown in either Figure 1(b) or (c), we can express the equivalence between the different selection ordering discussed above in the form of an equation, such as the following.

$$[\![e]\!](Factor.three)(Input.x) = [\![e]\!](Input.x)(Factor.three)$$

The choice calculus is about declaring dimensions on the one hand and selecting tags on the other hand. The original choice calculus does not support these two aspects in a balanced way. Dimension declaration is supported syntactically by expressions of the form **dim** $D\langle t,\ldots,t\rangle$ **in** e, but tag selection is only supported semantically by the notion of tag selection that is external to the calculus itself. We would like to complement the semantic account for tag selection with a syntactic account. That requires two steps: (1) a syntactic form to request tag selection as part of a choice calculus expression, and (2) a contraction law to connect tag selection with dimension declaration syntactically. By explaining dimension declaration and tag selection as the corresponding introduction and elimination forms with a contraction law, we provide a symmetric treatment of these two complementary constructs, which makes it more similar to the treatment as found in the λ-calculus. This analogy helps us to better understand the current and desired treatment of opperations in the choice calculus.

language	domain	mechanism	introduction form	elimination form	reference form
choice calculus	variation	selection $\lfloor e \rfloor_{D.i}$	dimension declaration **dim** $D\langle t,\ldots,t\rangle$ **in** e	tag selection N/A	choice $D\langle e,\ldots,e\rangle$
λ-calculus	computation	substitution $e[x \mapsto e]$	functional abstraction $\lambda x.e$	application $e\ e$	variable x

(a) Comparison

$$(\lambda x.e_1)\ e_2 \equiv e_1[x \mapsto e_2]$$

(b) β-equivalence.

$$\lambda x.(e\ x) \equiv e$$

(c) η-equivalence.

Figure 3: Overview of the choice calculus and its analogy with the λ-calculus.

2.3 Relationship to the Lambda Calculus

The λ calculus models computation by substitution. It features three syntactic forms: $\lambda x.e$ to introduce functions, applications $e\ e$ to eliminate functions, and variable occurrences x that refer to variable names bound by λ. Substitution $e[x \mapsto e]$ is a meta-function that eliminates variable occurrences by replacing them with some other term. The expressive power of the λ-calculus arises from the interaction of introduction and elimination forms as specified by β and η equivalence:

- β-equivalence specifies that an λ-abstraction nested on the left-hand side of an application triggers substitution, see Figure 3(b). In the practice of functional programming, β-equivalence corresponds to inlining of function definitions. In the theory of computation, β-reduction is used to model step-wise computation of the value of an expression.
- η-equivalence specifies that an application to a variable nested in the λ-abstraction of that variable is superfluous, see Figure 3(c). In the practice of functional programming, η-expansion can be used to control evaluation order. In the theory of computation, η-equivalence is used to model extensional equivalence.

The relationship of the choice calculus to these concepts is shown in Figure 3(a). Just like the λ-calculus models computation by substitution, the choice calculus models variation by tag selection, so substitution corresponds to selection. The choice calculus features dimension declarations that correspond to λ-expressions and choice nodes that correspond to variable occurrences. There is no equivalent of application, however, and therefore no rules that correspond to β and η equivalence. By not having an elimination form, that is, selection as a *syntactic* construct, the choice calculus misses out on such powerful interactions.

3. INTERNALIZING SELECTION

We propose to add a syntactic form for selection to the choice calculus: **select** $D.t$ **from** e. The full syntax of the choice calculus with selection is shown in Figure 4. Intuitively, an expression of the form **select** $D.t$ **from** e should mean the same as e after selecting $D.t$. That is, we expect the new semantics $[\![\cdot]\!]$ to relate to the old semantics $[\![\cdot]\!]_{old}$ in the following way.

$$[\![\textbf{select}\ D.t\ \textbf{from}\ e]\!] = [\![e]\!]_{old}(D.t)$$

Therefore, the following expression e_{ab} should be equivalent to `1`.

$$e_{ab} = \textbf{select}\ D.a\ \textbf{from}\ \textbf{dim}\ D\langle a,b\rangle\ \textbf{in}\ D\langle \texttt{1},\texttt{2}\rangle$$

In the following we present some patterns that illustrate how the **select** construct increases the expressiveness of the choice calculus.

e	::=	$a\prec e,\ldots,e\succ$	*Object Structure*
	|	**dim** $D\langle t,\ldots,t\rangle$ **in** e	*Dimension*
	|	$D\langle e,\ldots,e\rangle$	*Choice*
	|	**select** $D.t$ **from** e	*Selection*
	|	**share** v = e **in** e	*Sharing*
	|	v	*Reference*

Figure 4: Choice calculus syntax.

3.1 Renaming

In the choice calculus as introduced in Section 2, a dimension cannot be renamed after it has been declared. This inflexibility can lead to invasive changes if existing code is adapted to work with a new feature model, that is, if code is reused in a different project or if the dimension names change due to evolution of the requirements. The lack of renaming also affects the potential for reusing independently developed code that might inadvertently use the same dimension names. In the choice calculus as-is, clashing dimensions names are allowed, but they are resolved in a fixed left-to-right order. This fixed order of configuration cannot suit the partial-configuration needs of all projects. A way to explicitly rename dimensions would give control back to the developers.

Tag selection can be used for renaming of dimensions. For example, if e exposes a dimension A with tags b and c, but we would rather expose a dimension X with tags y and z, we can wrap the expression e as follows:

share v = e **in**
  dim $X\langle y,\ z\rangle$ **in**
    $X\langle$**select** $A.b$ **from** v, **select** $A.c$ **from** $v\rangle$

This construction is not invasive: The implementation of e remains unchanged. This is similar to η-expansion in the λ-calculus.

3.2 Modularity

The choice calculus without a syntactic form for tag selection cannot fully model nested product lines, that is, variational software artifacts that reuse other variational software artifacts as components. The missing piece is fine-grained control over how the variability of a component contributes to the variability of the overall system. We distinguish three cases:

- *Direct contribution* to the overall variablity, that is, the dimension of the component is automatically exposed to clients of the overall system. This is the default behavior of the choice calculus for nested dimension declarations.
- *Local resolution*, that is, the dimension is fully configured as an implementation detail of the component that uses it, and its existence is kept unknown to clients of the overall system to ensure proper information hiding. This is trivially supported by wrapping the component in a **select** form.
- *Indirect contribution*, that is, the dimension of the component is explicitly mapped to one or more dimensions of the

overall system. The complexity of the mapping can range from simple renaming to complex dependencies on several dimensions. This is supported by wrapping the component in dependent selections as in Section 3.1.

The situation is similar to how the λ-calculus supports combinator libraries (collections of closed λ-terms) and their adaption to given interfaces by wrapper code. We believe that support for code reuse is crucial for scaling a calculus to large systems, and a syntactic form for selection can provide that support.

3.3 Operational Semantics

The semantics of the choice calculus without tag selection are specified using a denotational approach. With the addition of a syntactic form for tag selection, we can hope to complement this denotational semantics with an operational semantics that is based on a contraction law for the **select** and **dim** forms. In the beginning of this section, we have shown a simple instance of such a law where the expression e_{ab} reduces to `1` because the elimination form **select** $D.a$ matches the introduction **dim** $D\langle a,b\rangle$. For a full operational semantics, we expect an equivalence like the following ($\lfloor e \rfloor_{D.i}$ selects the ith alternative of all free D choices in e [6]).

$$\textbf{select } D.t_i \textbf{ from } (\textbf{dim } D\langle t_1, \ldots, t_i, \ldots, t_n\rangle \textbf{ in } e) \equiv \lfloor e \rfloor_{D.i}$$

This rule triggers selection just as β-reduction triggers substitution in the λ-calculus, see Figure 3(b). Additionally, we would have to add a contraction for **share** v and v to define the semantics of sharing, and congruence rules for other pairs of syntactic forms.

It turns out, however, that the intended meaning of selection is less clear than we hoped, so we have to refrain from defining a full semantics in this paper. Instead, we use example expressions involving **select** to discuss trade-offs for such a semantics.

4. OPEN QUESTIONS

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

4.1 Undeclared Dimension

The first question is what to do when a dimension that is selected is not declared, as in the following example.

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

There are two possible definitions for the selection in an undeclared dimension.

1. It is ill-formed, and an error is reported.
2. It is idempotent. The example is equivalent to `1 + 2`.

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

4.2 Multiple Dimensions

The meaning of a selection is also unclear when there are multiple matching declarations in parallel. This situation is encountered when dimension declarations are nested in different siblings of a program structure, as in the following example.

select $D.a$ **from**
(**dim** $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$) `+` (**dim** $D\langle a,c\rangle$ **in** $D\langle 3,4\rangle$)

There are at least four possible resolutions:

1. Selections can only be applied directly to dimension declarations, preventing the multiple dimension issue from arising. The example is ill-formed and an error is reported.
2. Selections that match multiple dimension declarations are considered ambiguous. The example is ill-formed and an error is reported.
3. The left-most matching dimension declaration is resolved. The example is equivalent to `1 +` (**dim** $D\langle a,c\rangle$ **in** $D\langle 3,4\rangle$). We call this behavior of eliminating one dimension with one selection *linear selection*. This selection behavior is used in the old semantics.
4. All matching dimension declarations are resolved. The example is equivalent to `1 + 3`. We call this behavior of removing all matching, parallel dimension declarations with one selection *comprehensive selection*.

The first resolution initially seems quite restrictive, although it emphasizes the correspondence between the choice calculus and lambda calculus, sketched in Section 2.3. Just as lambda abstractions are reduced only when applied directly to an argument, dimensions are only eliminated when directly affected by a selection. This may help to enforce modularity since only a dimension declaration at the root of an expression can be referenced and eliminated. Section 6 describes a type system that enforces this constraint.

A simple observation is that the linear and comprehensive forms of selection are equivalent when all of the dimensions declared in an expression use different dimension names. An expression that satisfies this constraint is called *dimension linear* [6]. In the absence of dimension linearity, however, there are many interesting trade-offs between the two approaches. An advantage of the comprehensive approach is that it suggests the following confluence relation.

$$\begin{aligned}&\textbf{select } D.t \textbf{ from } a\langle\!\prec e_1,\ldots,e_n\succ\!\rangle\\ \equiv\ & a\langle\!\prec \textbf{select } D.t \textbf{ from } e_1,\ldots,\textbf{select } D.t \textbf{ from } e_n\succ\!\rangle\end{aligned}$$

This relationship can be used to push selection operations down to the dimensions they affect, leading to a simple, purely syntactic account of selection. Here we encounter our first interaction with a resolution for another challenge. If some e_i does not contain a dimension declaration for D, we encounter the undeclared dimension challenge described in Section 4.1. To preserve the confluence relation, we will prefer that selections without corresponding dimension declarations are idempotent.

While the comprehensive solution supports pushing selections down in an expression, the linear approach better supports lifting them up. Consider the following expression.

$$a\langle\!\prec e_1,\ldots,\textbf{select } D.t \textbf{ from } e_i,\ldots,e_n\succ\!\rangle$$

With the comprehensive approach, the selection can only be factored out of the object structure if $e_1,\ldots,e_{i-1},e_{i+1},\ldots,e_n$ do not contain an unselected dimension D, which would be captured. With the linear approach, we can factor the selection out by prioritizing e_i over $e_1,\ldots,e_{i-1}$ by introducting a sharing construct.

select $D.t$ **from**
share $v = e_i$ **in** $a\langle\!\prec e_1,\ldots,v,\ldots,e_n\succ\!\rangle$

With the linear approach, a selection can be pushed down an object structure only by examining each of the subexpressions to determine which contains the dimension that would be matched. This is undesirable since it is not a local syntactic transformation [7].

The choice of comprehensive or linear selection interacts with many of the other challenges describes in the rest of this section.

4.3 Undeclared Tag

What do we do when the dimension referred to in a selection is declared, but it does not contain the selected tag? This is illustrated in the following example.

select $D.a$ **from**
  dim $D\langle b,c\rangle$ **in** $D\langle 1,2\rangle$

Assuming we consider selections in undeclared dimensions to be idempotent, there are two ways to view mismatched tag names:

1. All unmatched selections should be treated similarly. Therefore, the selection is idempotent and the example is equivalent to **dim** $D\langle b,c\rangle$ **in** $D\langle 1,2\rangle$.
2. A selection of an undeclared tag in a declared dimension is considered ill-formed. An error is reported.

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

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

If we assume that the selection does not match the left dimension declaration and does not induce an error, then the expression exhibits a strange asymmetry with both linear and comprehensive selection: With linear selection, observe that we can select either $D.b$ or $D.c$ from the left dimension, or $D.a$ from the right dimension, but not $D.b$ from the right dimension. With comprehensive selection, if we choose $D.b$, the two dimensions will be synchronized as expected, but if we first choose $D.a$ or $D.c$, we can then unilaterally choose $D.b$ in the remaining dimension. Since this is probably not the intended behavior, we might choose the second solution, which prevents this scenario by making the top-level selection of $D.a$ or $D.c$ an error. However, this effectively restricts all parallel dimension declarations to define exactly the same tags in the context of comprehensive selection, so this may be an overly strict requirement.

4.4 Dependent Dimensions

Another question is how to treat the selection of a dimension that is declared nested within a choice in a different dimension. Such nested dimensions are said to be *dependent* on the corresponding tags in the outer dimension(s). In the following example, the dimension B is dependent on $A.a$.

select $B.c$ **from**
  dim $A\langle a,b\rangle$ **in**
    $A\langle$**dim** $B\langle c,d\rangle$ **in** $B\langle 1,2\rangle, 3\rangle$

There are at least three possible resolutions.

1. Selection from dependent declarations is considered ill-formed. (The appropriate selection in the outer dimension must be applied first.) An error is reported.
2. The matching declaration is resolved, preserving alternatives where the selected dimension is not declared. The example is equivalent to **dim** $A\langle a,b\rangle$ **in** $A\langle 1,3\rangle$.
3. The matching declaration is resolved, removing alternatives where the selected dimension is not declared. The expression is equivalent to **dim** $A\langle a\rangle$ **in** $A\langle 1\rangle$.

An argument for the third solution can be made from a usability perspective: Making a selection in a dependent dimension requires the selection of the tags it is dependent on. Consider the following excerpt from a variational list of errands.

select *BuyPie.yes* **from**
  dim *VisitBakery*$\langle$*yes*,*no*$\rangle$ **in**
    VisitBakery$\langle$
      dim *BuyPie*$\langle$*yes*,*no*$\rangle$ **in**
        BuyPie$\langle$buy pie, leave bakery$\rangle, \ldots\rangle$

If we decide to buy the pie, then we must necessarily visit the bakery, so we can consider the above expression equivalent to the following, in which we are now forced to select *yes* in the *VisitBakery* dimension.

dim *VisitBakery*$\langle$*yes*$\rangle$ **in**
  VisitBakery$\langle$buy pie$\rangle$

We could consider the selection of a dependent dimension to directly imply the selection of the tags it is dependent on, so that the above example is equivalent to simply "buy pie". However, this approach does not scale to the case when a dependent dimension occurs in more than one alternative of the surrounding choice.

The drawback of this automatic reduction of variability is that it is quite complicated and perhaps better described as a combination of more primitive operations. The other approaches are simpler.

4.5 Scope of Selection

The final challenge is to determine whether selection should be lexically or dynamically scoped. This is relevant since dimensions can be declared in shared expressions and reused. This is a divergence from our previous work, where we have considered shared expressions to be expanded only after all dimensions have been eliminated [6]. To illustrate, consider the following expressions:

(a) **share** v = (**dim** $A\langle a,b\rangle$ **in** $A\langle 1,2\rangle$) **in** v+v
(b) **dim** $A\langle a,b\rangle$ **in** $A\langle$1+1,2+2$\rangle$
(c) (**dim** $A\langle a,b\rangle$ **in** $A\langle 1,2\rangle$)+(**dim** $A\langle a,b\rangle$ **in** $A\langle 1,2\rangle$)

In the context of $[\![\cdot]\!]_{Old}$, (a) and (b) are equivalent. However, one motivation for a syntactic representation of selection is to support the reuse and configuration of variational modules. Therefore, in the context of $[\![\cdot]\!]$, we will consider (a) and (c) to be equivalent.

The issue of the scope of the selection construct can now be seen in the following example:

share v = (**dim** $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$) **in**
  select $D.a$ **from** v

There are two possible resolutions:

1. Selection has lexical scope. The example is equivalent to **share** v = (**dim** $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$) **in** v.
2. Selection has dynamic scope. The example is equivalent to **share** v = 1 **in** v.

Lewis et al. [15] propose a type system for dynamically scoped variables that is based on a similar idea. To reuse a variational expression, and configure it differently in different locations, seems to require a dynamically scoped selection operation. For example, the following would be equivalent to 1+2.

share v = (**dim** $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$) **in**
  (**select** $D.a$ **from** v) + (**select** $D.b$ **from** v)

However, dynamic scoping is risky since dimensions can be unexpectedly captured by selections. This is similar to the issue of macro hygiene in metaprogramming systems [14]. For example, in the following expression where v is defined far away from the selection, the user likely intended the selection to affect only the second declaration of D.

share v = (**dim** $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$) **in**
...
select $D.a$ **from**
v + (**dim** $D\langle a,c\rangle$ **in** $D\langle 3,4\rangle$)

This expectation, which is fulfilled by lexically scoped selection, leads to the following equivalent expression.

dim $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$ + 3

However, with dynamic scoping, the dimension in the bound expression v will be captured by the selection. The meaning now depends on whether we choose the linear or comprehensive form of selection, described in Section 4.2. Under comprehensive selection, the example is equivalent to 1+2. With linear selection, the intended dimension is not selected at all, and the example is equivalent to the following expression.

1 + **dim** $D\langle a,c\rangle$ **in** $D\langle 3,4\rangle$

In both cases, the unwanted dimension capture can be avoided by simply localizing the selection to the intended dimension, such that the variable is no longer in its scope. Thus, we can rewrite our original example in the following way, which behaves as expected.

share v = (**dim** $D\langle a,b\rangle$ **in** $D\langle 1,2\rangle$) **in**
...
v + (**select** $D.a$ **from dim** $D\langle a,c\rangle$ **in** $D\langle 3,4\rangle$)

While unexpected dimension captures are bad, dynamically scoped selection is a powerful tool for the reuse of variational expressions. Consider the following expression in which the expression e_{big} is very large and declares a dimension **dim** $D\langle a,b\rangle$.

share $v = e_{big}$ **in**
(**select** $D.a$ **from** v) + (**select** $D.b$ **from** v)

Without dynamic scoping, e_{big} cannot be reused and configured separately in different parts of the program. This provides a practical motivation for dynamically scoped selection.

5. CONFIGURATION SAFETY

We next discuss the design of a type system for the choice calculus whose purpose is, first, to ensure choices in expressions are properly bound by dimension declarations and, second, to track the configuration status of expressions. The type system is speculative since we have not yet decided on the semantics definition. Note that this kind of type system is different in purpose and structure from other variation type systems that try to ensure type safety of object languages in the presence of variation, such as [2] or [13].

Configuration information about choice calculus expressions is captured by a judgment of the form: $\Gamma \vdash e : \Delta$. Here, e is a choice calculus expression, Δ is its *configuration type*, and Γ is an *environment* to store unbound choices and the configuration type of shared variables e. The configuration type Δ of an expression describes the dimensions we have to configure in order to generate a variant. We represent the configuration type by a set of dimension and tag specifications, that is, $\Delta = \{d_1,\ldots,d_n\}$ where each d_i is of the form $D_i\langle t_1^i,\ldots,t_{n_i}^i\rangle$. Expressions with configuration type $\{\}$ are fully configured. Note that this flat set representation requires that expressions are dimension linear (all contained dimension declarations use different names, see Section 4.2). This is not a serious limitation since we can always rename dimension declarations (see also [2]). We write $\Delta \oplus \Delta'$ for the union of the configurations Δ and Δ' that is defined only if Δ and Δ' are disjoint. (We also occasionally omit the set brackets from singleton sets and write $d \oplus \Delta$.)

Unbound choices are represented in the environment Γ by the name of the choice's dimension and the number of its alternatives, written as $D : n$, and configuration assumptions for shared variables are written as $v : \Delta$.

$$\Gamma \quad ::= \quad \varnothing \mid \Gamma,\, D : n \mid \Gamma,\, v : \Delta$$

The typing judgment $\Gamma \vdash e : \Delta$ captures two important properties about choice calculus expressions.

- An expression e is *well formed* [6] if it does not contain unbound choices, and all choices have the correct number of alternatives. This is the case if $\varnothing \vdash e : \Delta$.
- An expression e is *fully configured* (or *plain* [6]) if e does not contain any dimension declarations for which a selection decision remains to be made. This is the case if $\Gamma \vdash e : \{\}$.

Of course, we can take these two properties together and say that an expression is well formed and fully configured if $\varnothing \vdash e : \{\}$.

In Figure 5 we present a set of rules that define the typing judgment in a conservative way, which makes specific assumptions and also imposes various restrictions on choice calculus expressions. We will discuss the implications of these decisions, and potential generalizations, alongside the explanation of the rules.

The OBJ rule collects the configuration requirements of all subexpressions and combines them into one requirement for the whole expression. Since Δ and Δ' must be disjoint, the type system assumes dimension linearity.

The rules for sharing and variable reference SHARE and REF are straightforward and similar to those in lambda calculus.

The CHOICE rule requires a corresponding dimension entry in the environment, which is later discharged by the DIM rule. Note that all alternatives must have the same requirements, this means we cannot type dependent dimensions (see Section 4.4). When the requirements of all alternatives are the same, any nested dimensions can always be factored out of the choice [6]. This restriction is necessary to support the flat structure of configuration types.

Finally, the SELECT rule removes a configuration obligation from the configuration type. For dimension-linear expressions, each **select** operation will remove one dimension declaration, so a well-formed expression e with $\varnothing \vdash e : \Delta$, requires exactly $|\Delta|$ selections to obtain a plain, fully configured expression.

The dimension linearity restriction can be lifted if we employ a more structured form of configurations types, where a tag in a dimension can lead to further required configuration.

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

We abbreviate $t \Rightarrow \{\}$ as simply t. Additionally, we associate a selection assumption i with dimensions in the environment Γ.

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

With these provisions, the CHOICE rule can be extended as follows.

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

That is, the type of a choice is the type of the alternative that is assumed to be selected. The DIM rule is extended to introduce selection assumptions and build the corresponding required decision.

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

Finally, the SELECT rule replaces a required decision with the type of the selected alternative.

$$\frac{\Gamma \vdash e' : \Delta' \qquad \Gamma,\, v : \Delta' \vdash e : \Delta}{\Gamma \vdash \textbf{share}\ v = e'\ \textbf{in}\ e : \Delta}\textsc{Share} \qquad \frac{v : \Delta \in \Gamma}{\Gamma \vdash v : \Delta}\textsc{Ref} \qquad \frac{\Gamma,\, D : n \vdash e : \Delta}{\Gamma \vdash \textbf{dim}\ D\langle t_1, \ldots, t_n\rangle\ \textbf{in}\ e : D\langle t_1, \ldots, t_n\rangle \oplus \Delta}\textsc{Dim}$$

$$\frac{\Gamma \vdash e_1 : \Delta_1 \quad \ldots \quad \Gamma \vdash e_n : \Delta_n}{\Gamma \vdash a\text{-}\prec e_1, \ldots, e_n \succ\text{-} : \Delta_1 \oplus \ldots \oplus \Delta_n}\textsc{Obj} \qquad \frac{D : n \in \Gamma \qquad \Gamma \vdash e_1 : \Delta \quad \ldots \quad \Gamma \vdash e_n : \Delta}{\Gamma \vdash D\langle e_1, \ldots, e_n\rangle : \Delta}\textsc{Choice} \qquad \frac{\Gamma \vdash e : D\langle t_1, \ldots, t_n\rangle \oplus \Delta \qquad t_i = t}{\Gamma \vdash \textbf{select}\ D.t\ \textbf{from}\ e : \Delta_n}\textsc{Select}$$

Figure 5: Configuration typing rules.

$$\frac{\Gamma \vdash e : \Delta \oplus D\langle t_1 \Rightarrow \Delta_1, \ldots, t_n \Rightarrow \Delta_n\rangle \oplus \Delta' \qquad t_i = t}{\Gamma \vdash \textbf{select}\ D.t\ \textbf{from}\ e : \Delta \oplus \Delta_i \oplus \Delta'}\textsc{Select}$$

If we assume that the first matching required decision is eliminated, this type system corresponds to the linear selection semantics.

6. MODULARITY

When considering nested software product lines, and hence software product lines as components of bigger product lines, the classical questions of information hiding and robustness with regard to evolution arise. From this point of view, it is important that the type of an expression is a proper abstraction of the expression which does not reveal too many (fragile) details of its structure.

In this light, the OBJ typing rule from Figure 5 is dangerous since it merges the configuration options of its subexpressions. Changing these subexpressions changes the type of the compound expression. In the linear semantics, the structure of a compound expression is also relevant for the semantics of the selection operation.

To address this problem, we present an alternative version of the choice calculus, the *modular choice calculus* (MCC), which is more restrictive with regard to nesting dimension declarations, but is more suitable for modular software product line components.

Figure 6 shows the syntax and typing rules for MCC (only the parts that differ from Figures 4 and 5 are given). The main difference in syntax is that multiple dimensions can be declared at once. The reason for this change is that, in MCC, the following two expressions have two very different meanings.

$$e_{AB} = \textbf{dim}\, A\langle\ldots\rangle, B\langle\ldots\rangle\ \textbf{in}\ \ldots$$
$$e_{A_B} = \textbf{dim}\, A\langle\ldots\rangle\ \textbf{in dim}\, B\langle\ldots\rangle\ \textbf{in}\ \ldots$$

The expression e_{AB} declares a product line with two configuration options, whereas e_{A_B} declares a product line with a single configuration option which, when selected, yields a product line with a single configuration option. Similar to the lambda calculus, nested product lines as in the former case must be configured one by one, starting from the outermost. This is why it is crucial that one can declare multiple dimension options at once, as in e_{AB}, because such product lines can be partially configured in arbitrary order.

MCC is also similar to the lambda calculus with regard to its operational semantics. In both calculi, introduction and elimination forms must be directly adjacent (possibly after some reduction) to trigger a contraction rule. In contrast, in the calculus described in the previous sections, the introduction and elimination forms for variability can have "remote" effects; they are propagated along the syntax tree. In terms of the discussion in Section 3.2, the modular variant only allows local resolution and indirect contribution.

While we don't show the semantics of the calculus here this design decision is reflected in the type system in Figure 6. As reflected in the OBJ and CHOICE rules, a compound term must consist of fully configured subterms. The SELECT rule allows partial configuration of a product line; each **select** eliminates the corresponding dimension from the type. Not shown is the special case when the last dimension is eliminated from the configuration type, in which case, the resulting configuration type would be $\{\}$. The **close** form serves to confirm that a product line has been fully configured.

Pragmatically, the main difference between the non-modular version of the calculus and MCC can be observed in an expression that contains subexpressions that are not fully configured. In MCC, the nested dimensions must be explicitly mapped to dimensions at the top level. For example, in the following expression, the nested dimension B is mapped to the top-level dimension A.

dim $A\langle a, b\rangle$ **in**
  ...
  share v = (**dim** $B\langle c, d\rangle$ **in** ...) **in**
    $A\langle$**select** $B.c$ **from** v, **select** $B.d$ **from** $v\rangle$

In the non-modular calculus, open configuration options of nested expressions are directly selectable from the top level. By requiring this explicit mapping we provide an abstraction boundary that prevents the propagation of evolutionary changes in a subexpression to the compound expression.

7. RELATED WORK

The C Preprocessor (CPP) [8] is a widely used annotative variation tool that also provides syntactic support for selection. A loose correspondence between concepts in the choice calculus and CPP is summarized in the following table.

Choice Calculus	C Preprocessor
$a\text{-}\prec e_1, \ldots, e_n \succ\text{-}$	Plain text
Tags: t, u	Macros: `T`, `U`
dim $D\langle t, u\rangle$	N/A
$D\langle e_1, e_2\rangle$	`#if T` / e_1 / `#elif U` / e_2 / `#endif`
select $D.t$	`#define T 1`

Unlike CPP, the choice calculus respects the AST of the underlying artifact. This is also a feature of other structured annotative variation tools, such as CIDE [11]. A configuration is identified in the choice calculus by a particular selection of tags, and in CPP, by a particular definition of the macros named in conditional compilation directives. Dimensions impose a structure on the configuration space that is not present in CPP, and choices ensure that this structure is respected at each variation point. External representations like feature models [10] and the Linux Kconfig tool [17] can fulfill a similar role as dimensions, but the consistent usage of macros in CPP is not enforced, which leads to bugs in practice [18].

Both the choice calculus and CPP support external configuration. In CPP, this corresponds to setting macros at the command line or in a Makefile. Like our new **select** construct, the CPP `#define` directive provides syntactic support for configuration from within the language itself. However, `#define` is much more complicated than **select** since (1) macros can be conditionally defined, (2) macros can be defined in terms of other macros, and (3) the value of a macro can change over the course of a program. This complexity makes it difficult to reason about variability in CPP-annotated code [12]. The extension we propose to the choice calculus provides a simpler, more structured view of configuration. This is reflected in the relatively simple type systems described in Sections 5 and 6.

$\delta ::= \overline{D\langle t, \ldots, t\rangle}$ *Dimension(s)*

$e ::= \ldots$
| **dim** δ **in** e *Declaration*
| **close** e *Selection Completion*

$\Delta ::= \{\}$ *Fully Configured*
| $\delta \Rightarrow \Delta$ *Required Decision*

$$\frac{\Gamma \vdash e_1 : \{\} \quad \ldots \quad \Gamma \vdash e_n : \{\}}{\Gamma \vdash a\prec e_1, \ldots, e_n\succ : \{\}}\textsc{Obj} \qquad \frac{D : n \in \Gamma \qquad \Gamma \vdash e_1 : \{\} \quad \ldots \quad \Gamma \vdash e_n : \{\}}{\Gamma \vdash D\langle e_1, \ldots, e_n\rangle : \{\}}\textsc{Choice}$$

$$\frac{\Gamma \vdash e : \{\} \Rightarrow \Delta}{\Gamma \vdash \textbf{close}\ e : \Delta}\textsc{Close} \qquad \frac{\Gamma, D_1 : n_1, \ldots, D_k : n_k \vdash e : \Delta \qquad \delta = \ldots, D_i\langle t_1^i, \ldots, t_{n_i}^i\rangle, \ldots}{\Gamma \vdash \textbf{dim}\ \delta\ \textbf{in}\ e : \delta \Rightarrow \Delta}\textsc{Dim}$$

$$\frac{\Gamma \vdash e : \delta \Rightarrow \Delta \qquad D\langle t_1, \ldots, t_n\rangle \in \delta \qquad \delta' = \delta \setminus \{D\langle t_1, \ldots, t_n\rangle\} \qquad t_i = t}{\Gamma \vdash \textbf{select}\ D.t\ \textbf{from}\ e : \delta' \Rightarrow \Delta}\textsc{Select}$$

Figure 6: Syntax and typing rules for modular choice calculus.

Some software product line systems provide language support for configuring individual products, such as the feature algebra supported by the AHEAD tool suite [1]. However, this language is purely external, and therefore more closely related to Make and KConfig than to the **select** construct described here.

While rare in implementation-level languages, internal language support for selection has been explored at the modeling level. Reiser has extended feature models with *configuration links* that support the reuse and (partial) configuration of feature models [16]. A configuration link maps a decision in the source feature model to a set of decisions in the target feature model, by way of a set of rules. This can be used to simulate the modeling of nested product lines, as described in Section 3.2. Similarly, Haber et al. present a meta model for describing variability in hierarchical, component-based systems, that contains constructs for locally configuring the variability in components [9].

More research has been done on external configuration at the level of feature models. For example, Czarnecki et al. enumerate the ways variation can be reduced in cardinality-based feature models [4]. This effectively yields a catalog of potential configuration operations, of which our **select** construct is just a single instance. Subsequently, Classen et al. have provided a formal semantics for the staged configuration of feature models [3].

8. CONCLUSIONS

We have presented an extension of the choice calculus with a selection operation. This has raised a number of design questions regarding the semantics of selection itself and potential configuration and modularity type systems for the choice calculus. The different potential semantics of selection can be primarily distinguished in two dimensions: (a) the scope of the sharing construct (static vs. dynamic) and (b) the reach of the selection operation (linear vs. comprehensive dimension elimination). Moreover, we have seen that the structure of the type system has a significant impact on the expressiveness of the choice calculus. While dimension linearity is a non-critical restriction, a flat model of configuration types precludes dependent dimensions.

We know from the vast literature on the lambda calculus that there are different semantics and type systems for different purposes and with different strengths and weaknesses. For the choice calculus we obtain a similar picture. In future work we will continue investigating more expressive type systems and their interaction with the different semantics.

9. ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation under the grants CCF-0917092 and CCF-1219165 and by the European Research Council under the grant #203099. We thank the anonymous reviewers for their helpful comments and Sebastian Erdweg, Sven Apel, Christian Kästner and Erik Ernst for discussions about the choice calculus and the need for a **select** form.

10. REFERENCES

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

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

[3] A. Classen, A. Hubaux, and P. Heymans. A Formal Semantics for Multi-Level Staged ConïňAguration. In *Work. on Variability Modeling of Software-Intensive Systems*, pages 51–60, 2009.

[4] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–168, 2005.

[5] M. Erwig. A Language for Software Variation. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*, pages 3–12, 2010.

[6] 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.

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

[8] GNU Project. *The C Preprocessor*. Free Software Foundation, 2011. `http://gcc.gnu.org/onlinedocs/cpp/`.

[9] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, and F. van der Linden. Hierarchical Variability Modeling for Software Architectures. In *Software Product Line Conf.*, 2011.

[10] 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.

[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, P. G. Giarrusso, and K. Ostermann. Partial Preprocessing C Code for Variability Analysis. In *Work. on Variability Modeling of Software-Intensive Systems*, pages 127–136, 2011.

[13] A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Int. Work. on Feature-Oriented Software Development*, pages 25–32, 2010.

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

[15] J. R. Lewis, J. Launchbury, E. Meijer, and M. B. Shields. Implicit Parameters: Dynamic Scoping with Static Types. In *ACM Symp. on Principles of Programming Languages*, pages 108–118, 2000.

[16] M.-O. Reiser. *Managing Complex Variability in Automotive Software Product Lines with Subscoping and Configuration Links*. PhD thesis, Technische Universität Berlin, Feb. 2009.

[17] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The Variability Model of the Linux Kernel. In *Work. on Variability Modeling of Software-Intensive Systems*, pages 45–51, 2010.

[18] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *GPCE Work. on Feature-Oriented Software Development*, pages 81–86, 2009.