REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*

2. REPORT TYPE

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:

a. REPORT

b. ABSTRACT

c. THIS PAGE

17. LIMITATION OF ABSTRACT

18. NUMBER OF PAGES

19a. NAME OF RESPONSIBLE PERSON

19b. TELEPHONE NUMBER *(include area code)*

AFOSR Final Performance Report

Project Title: **A Theory Of Structured Change**

Grant Number: **FA9550-09-1-0229**

Principal Investigator: Martin Erwig
School of Electrical Engineering and Computer Science
Oregon State University
erwig@eecs.oregonstate.edu

Period of Report: February 2009 – March 2012

1 Project Objectives

The overall goal of this research project is to develop a theoretical framework for describing and reasoning about changes in software. This theory should provide a basis for the implementation of tools to support software maintenance tasks in the presence of changes. For the development of this theory the following specific research objectives have been identified.

(1) *Representation of changes and changeable objects.* Find a simple but flexible and expressive representation for objects and their changes, and provide a formal definition for the semantics of the change representation.
(2) *Properties of change representations.* Identify laws for the change representation that characterize the principle nature of changes and can serve as the basis for identifying semantics-preserving change transformations.
(3) *Property preservation.* One principal problem associated with changes is that they might invalidate important properties of the object to be changed. Therefore, we need to find conditions that guarantee the preservation of these properties under changes, or more generally, the evolution of properties along well defined gradients.
(4) *Efficient property checking.* We need efficient ways to check properties for change objects since the representation of changes within objects will produce many different objects at once.
(5) *Change editing and exploration.* Changes transform changeable objects into one another. A collection of changeable objects and their relating changes can be viewed as a space that can be navigated. We want to know how change exploration can be supported by operations that could form the basis for a corresponding editing tool, and how property preservation and evolution can be integrated into this process.

2 Status of Effort

This project has made substantial progress toward the goals and objectives formulated in the project proposal. In this section we will briefly summarize the major accomplishments that we have made. In Section 3 we will then describe the most important findings and results in more detail.

A first important insight is that the concept of change can be regarded as a special form of variation. More specifically, if we identify one of two given variants A and A' as "new" (let's say A') and the other one as "old", then we can speak of a change "from" A "to" A'. In other words, the concept of *variation* is in a sense a symmetric generalization of the notion of change. In most situations it is not necessary to make this distinction between different roles of variants, and since all the theoretical results hold for variation in general, we will talk in the remainder of this document about variation instead of change. Since the purpose of having variation in software is to offer choices, we also call a group of variants a *choice*.

With regard to the first two objectives of finding a change representation and investigating its properties, we have developed the *choice calculus* [6], a formal approach to representing variation and changes in structured documents. The key constructs of the choice calculus are scoped choices that can be arbitrarily nested and grouped into dimensions. For dimensions and choices we have identified numerous laws and transformation rules that allow flexible representations for variation to support different editing and viewing scenarios. We have also developed a design theory that can improve representations by eliminating redundancies or dead variants.

The representation offered by the choice calculus has applications in many areas [4]. One specific

application is in the domain of product engineering. We have developed techniques to supply product engineers with tailored feature-selection sequences to derive products from (software) product lines more efficiently [1]. The method is based on the selectivity of features (that is, the number of products that contain a particular feature) and the impact of a selected feature on the selection of other features. Moreover, our approach helps with the problem of unexpected side effects of feature selection in later stages of the selection process, which is commonly considered a difficult problem. An evaluation of the technique demonstrates significant improvements in efficiency for the feature selection process.

The integrated representation of changes in software through choices, which becomes possible through the constructs of the choice calculus, has also led to the idea of program fields as an abstraction for representing sets of programs [5]. Program fields offer new possibilities for the systematic management of software changes.

We have addressed the third and fourth research objectives regarding property checking and preservation by developing a type system for variational lambda calculus, a variant of lambda calculus that integrates concepts of the choice calculus to represent variation in functional programs [3]. To make variational type systems feasible in practice, we have developed a method for variational type inference that determines the types for sets of related programs. Since type errors may be present only in some of the potentially large number of program variants, it is important to ensure that type information for type-correct variants is cleanly separated from the type errors of incorrect variants [2]. The brute-force approach of generating all variants and type checking each one individually is generally not feasible since the number of plain programs represented by a variational program grows exponentially with the number of dimensions of variation. The use of the choice calculus leads to significant improvements since it can exploit the fact that related programs contain many shared parts and that many of the types of even non-shared program parts are identical, which leads to a reduction of the number of choices in the computed types. This work has produced a variational unification algorithm and insights into algorithms on variational structures that transcend the particular application of type inference.

Regarding the objective of change editing we have performed a user study that has demonstrated that the dimension-structured choice constructs of the choice calculus helps users reason about variational code faster and more accurately and ultimately supports code understanding better than traditional CPP annotations [8].

Finally, to support the objective of exploring changes and change editing we have developed a domain-specific language (DSL) to express transformations of variation representations [7]. This DSL provides the basis for writing programs to query, manipulate, and analyze variation structures. These tasks are examples of the more general notion of *variation programming*. The concept of variation programming specifically supports the systematic migration of software and the exploratory editing of software artifacts. The developed DSL illustrates how variation programming tasks can be dealt with systematically. This DSL separates variation representation and transformation into two distinct levels, which limits its scope. Therefore, we have also developed a generalization of the choice calculus that includes computational features [9] that can arbitrarily mix and nest choice annotations and transformations.

3 Accomplishments & Findings

In this section we describe the individual accomplishments of the research project in more detail. The following accomplishments will be discussed.

1. The choice calculus as a representation for variation and change
2. A formal semantics for the choice calculus

3. Equivalence relationships for choice calculus expressions
4. Normalization of choice calculus expressions
5. Design rules for choice representations
6. Program fields for representing sets of related programs
7. Variation types
8. A variational type system
9. Variational type inference
10. Making variational type systems error-tolerant
11. Support for variational editing
12. A DSL for variation programming

3.1 The choice calculus representation

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

As an example, consider the following four implementations of the function `twice`, which takes a numerical argument and returns that value doubled.

		Implementation	
		plus	*times*

Name	x	`int twice(int x) {`
		`return x+x;`
		`}`
			`int twice(int x) {`
			`return 2*x;`
			`}`

	y	`int twice(int y) {`
		`return y+y;`
		`}`
			`int twice(int y) {`
			`return 2*y;`
			`}`

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

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

```
dim Par⟨x,y⟩ in
dim Impl⟨plus,times⟩ in
twice (int Par⟨x,y⟩) {
  return Impl⟨Par⟨x,y⟩+Par⟨x,y⟩,2*Par⟨x,y⟩⟩;
}
```

In this example, we begin by declaring the two dimensions of variation using the choice calculus **dim** construct. For example, **dim** $Par\langle x,y\rangle$ declares a new dimension *Par* with *tags* x and y, representing the two possible parameter names. The **in** keyword denotes the scope of the declaration, which extends to the end of the expression if not explicitly indicated otherwise (for example, by parentheses).

Each dimension represents an incremental decision that must be made in order to resolve a choice

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

Figure 1: *Choice calculus syntax. Arbitrary tree structures that store information of type "a" can be annotated by choices and dimensions. A sharing construct allows the removal of redundancy in variation representations.*

calculus expression into a concrete program variant. The choices bound to that dimension are synchronized with this decision. This incremental decision process is called *tag selection*. When we select a tag from a dimension, the corresponding alternative from every bound choice is also selected, and the dimension declaration itself is eliminated. For example, if we select the y tag from the *Par* dimension (*Par.y*), we produce the following choice calculus expression in which the *Par* dimension has been eliminated and each of its choices has been replaced by its second alternative.

```
dim Impl⟨plus,times⟩ in
twice (int y) {
  return Impl⟨y+y,2*y⟩;
}
```

If we then select *Impl.times*, we produce the variant of `twice` in the lower-right corner of the shown grid of variants.

In the above examples, the choice calculus notation is embedded within the syntax of the object language. This embedding is not a textual embedding in the way that, for example, the C Preprocessor's `#ifdef` statements are integrated with program source code. Instead, choices and dimensions operate on an abstract-syntax-tree view of the object language. For example, the AST for `twice x = x+x` can be written as `=`$\prec$`twice,x,+`$\prec$`x,x`$\succ\succ$, that is, the definition is represented as a tree that has the = operation at the root and three children, (1) the name of the function (`twice`), (2) its parameter (`x`), and (3) the defining expression, which is represented by another tree with root + and two children that are both given by `x`.

The syntax of choice calculus expressions follows from the discussion in the previous section and is provided explicitly in Figure 1. In addition to the constructs already discussed the choice calculus also offers a construct for sharing expressions (for details, see [6, 7]).

There are a few syntactic constraints on choice calculus expressions not expressed in the grammar. First, all tags in a single dimension must be pairwise different so they can be uniquely referred to. Second, each choice $D\langle e^n\rangle$ must be within the static scope of a corresponding dimension declaration $\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e$. That is, the dimension D must be defined at the position of the choice, and the dimension must have exactly as many tags as the choice has alternatives. Finally, each sharing variable reference v must be within scope of a corresponding **share** expression defining v.

3.2 Choice calculus semantics

In Section 3.1 we have described tag selection as a means to eliminate a dimension of variation. We write $\lfloor e\rfloor_{D.t}$ for the selection of tag t from dimension D in expression e. Tag selection consists of (1) finding the first declaration $\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e'$ in a preorder traversal of e, (2) replacing every choice bound

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

Figure 2: *Choice elimination. Selection of the nth tag from a dimension leads to the selection of the nth alternative from each choice bound by that dimension. Each such selection eliminates one dimension and all bound choices.*

---

by the dimension in e' with its ith alternative, where i is the index of t in t^n, and (3) removing the dimension declaration. Step (2) of this process is called *choice elimination*, written $\lfloor e'\rfloor_{D.i}$ (where the tag name has been replaced by the relevant index), and defined formally in Figure 2. This definition is mostly straightforward, replacing a matching choice with its ith alternative and otherwise propagating the elimination downward. Note, however, that propagation also ceases when a dimension declaration of the same name is encountered—this maintains the static scoping of dimension names.

We write $[\![e]\!]$ to indicate the semantics of choice calculus expression e, which is given by a function that maps sequences of tags to plain expressions. We represent the denotation of e (that is, the mapping from decisions to plain expressions) as a set of pairs, and we represent decisions as n-tuples of dimension-qualified tags. For simplicity and conciseness, we enforce in the definition of the semantics that tags are selected from dimensions in a fixed order, the order that the dimension declarations are encountered in a preorder traversal of the expression (see [6] for a discussion of this design decision). For instance, in the following example, tags are always selected from dimension A before dimension B.

$$
\begin{aligned}
&[\![\mathbf{dim}\ A\langle a_1,a_2\rangle\ \mathbf{in}\ A\langle 1,\mathbf{dim}\ B\langle b_1,b_2\rangle\ \mathbf{in}\ B\langle 2,3\rangle\rangle]\!] =\\
&\qquad\{(A.a_1,1),((A.a_2,B.b_1),2),((A.a_2,B.b_2),3)\}
\end{aligned}
$$

Note that dimension B does not appear at all in the decision of the first entry in this denotation since it is eliminated by the selection of the tag $A.a$.

The formal definition of the semantics of choice calculus expressions in terms of a helper function V is shown in Figure 3. The parameter to this function, ρ, is an environment, implemented as a stack, mapping **share**-variables to plain expressions. The semantics of e is then defined as an application of V with an initially empty environment, that is, $[\![e]\!] = V_{\varnothing}(e)$.

In the definition we use δ to range over decisions, concatenate decisions δ_1 and δ_2 by writing $\delta_1\delta_2$, and use δ^n to represent the concatenation of decisions $\delta_1,\ldots,\delta_n$. Similarly, lists of expressions e^n can be expanded to $e_1,\ldots,e_n$, and likewise for lists of tags t^n. We associate v with e in environment ρ with the notation $\rho\oplus(v,e)$, and lookup the most recent expression associated with v by $\rho(v)$.

For structure expressions there are two sub-cases to consider. If the expression is a leaf, then the expression is already plain, so the result is an empty decision (represented by the nullary tuple $()$) mapped to that leaf. Otherwise, we recursively compute the semantics of each subexpression and, for each combination of entries (one from each recursive result), concatenate the decisions and reconstruct the

$$
\begin{aligned}
V_\rho(a\text{-}\!\!\prec\!\succ\!\text{-}) &= \{((), a\text{-}\!\!\prec\!\succ\!\text{-})\} \\
V_\rho(a\!\prec\! e^n \!\succ) &= \{(\delta^n, a\!\prec\! e'^n\!\succ) \mid (\delta_1, e'_1) \in V_\rho(e_1), \ldots, (\delta_n, e'_n) \in V_\rho(e_n)\} \\
V_\rho(\textbf{dim}\ D\langle t^n\rangle\ \textbf{in}\ e) &= \{((D.t_i, \delta), e') \mid i \in \{1, \ldots, n\},\ (\delta, e') \in V_\rho(\lfloor e \rfloor_{D.i})\} \\
V_\rho(\textbf{share}\ v{=}e_1\ \textbf{in}\ e_2) &= \bigcup \{\{(\delta_1\delta_2, e'_2) \mid (\delta_2, e'_2) \in V_{\rho \oplus (v, e'_1)}(e_2)\} \mid (\delta_1, e'_1) \in V_\rho(e_1)\} \\
V_\rho(v) &= \{((), \rho(v))\}
\end{aligned}
$$

Figure 3: *Semantics of choice calculus expressions. The semantics is computed with the help of an auxiliary function V, which takes an environment of sharing definitions as an additional parameter. The semantics of an expression e is given by* $[\![e]\!] = V_\varnothing(e)$.

(now plain) structure expression.

On a dimension declaration, we select each tag t_i in turn, computing the semantics of $\lfloor e \rfloor_{D.i}$ and prepending $D.t_i$ to the decision of each entry in the result. Note that there is no case for choices in the definition of V. Since we assume that all choices are bound, all choices will be eliminated by selections invoked at their binding dimension declarations. In the event of an unbound choice, the semantics are undefined.

3.3 Equivalence relationships

The representation of variation with choices is not unique, that is, choices can be generally represented on different levels of granularity, and dimension definitions can be moved around too. For example, the following three expression are all equivalent in the sense that $[\![e]\!] = [\![e']\!] = [\![e'']\!]$.

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

Different representations are useful for different purposes. For example, maximally factored choices (as in e and e'') keep common parts out of alternatives as much as possible and thus simplify the editing of these common parts, avoiding update anomalies. On the other hand, fewer and bigger choices that repeat common parts are sometimes better suited to compare alternatives than a huge collection of fine-grained representations. Moreover, having dimensions as far at the top as possible (as in e and e') reveals the variational structure better than deeply nested dimensions. This might be desirable or not, depending on the context.

A complete set of equivalence relationships can be obtained by observing that in principle any syntactic form, that is, *Structure*, *Dimension*, *Choice*, *Sharing*, or *Reference*, can be commuted with any other. This complete set can be found in [6]. An excerpt of the relationships is shown in Figure 4, which shows rules for factoring and distributing choices across other syntactic constructs. In the rules we make use of a further notational convention to expose the ith element of a sequence. The pattern notation $e^n[i\!:\!e']$ expresses the requirement that e_i has the form given by the expression (or pattern) e'. For example, $e^n[i\!:\!e'+1]$ says that e_i must be an expression that matches $e'+1$.

In the case of nested choices for the same dimension D (rule C-C-MERGE), distribution amounts to merging the two choices into one. In that case the nested choice ($D\langle e'^n\rangle$) does not really present a choice

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

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

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

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

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

C-C-Merge
$D\langle e^n[i:e'_i]\rangle \equiv D\langle e^n[i:D\langle e'^n\rangle]\rangle$

Figure 4: *Choice commutation rules. Applied from right to left, these rules facilitate the factoring of common parts out of choices to make choices more focused. Applied from left to right, the rules distribute context into choices, which can be useful to increase the comprehensibility of individual variants and to support the editing of variants. This notion of factoring and distribution is also illustrated on a high level in Figure 5, shown on page 10.*

since all alternatives except e'_i are dead and cannot be reached because the semantics of tag selection recursively selects the same component from a nested choice. That is, if tag selection selects $D\langle e'^n\rangle$ as the ith alternative of the outer choice, it will also select e'_i from the inner one.

3.4 Variation Normal Forms

The rules presented in Section 3.3 can be used to transform expressions in many different ways. We have identified three strategically significant representations: *dimension normal form*, *choice normal form*, and *dimension-choice normal form*. We can show that any expression can be transformed into choice normal form, and that any "linearly dimensioned" expression (see below) can be transformed into dimension normal form and consequently, dimension-choice normal form.

We say that an expression e is in *choice normal form* (*CNF*) if it contains only choices that are maximally factored, that is, e is in CNF if no subexpression of e matches the right-hand side of any of the rules given in Figure 4 (without violating a premise). CNF is significant because it reduces redundancy in the representation. This is an important feature for the development of variation editing tools because it decreases the risk of update anomalies.

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

Finally, we say that an expression is in *dimension-choice normal form* (*DCNF*) if it is in choice normal form and in dimension normal form. Naturally, DCNF combines the benefits of both CNF and DNF, avoiding redundancy and clearly revealing the dimension structure. It is therefore a prime candidate for a variation representation in an editor tool or IDE since it avoids update anomalies while editing and

it groups related dimensions.

A choice calculus expression e is said to be *well dimensioned* if each choice $D\langle e_1,\ldots,e_n\rangle$ is in scope of a corresponding dimension definition $\mathbf{dim}\ D\langle t_1,\ldots,t_n\rangle$, that is, the dimension declaration that binds D introduces exactly as many tags as each choice that references D has alternatives. If all dimension names in e that are introduced by a **dim** construct are pairwise different, we say that e is *dimension linear*. We call an expression that is well dimensioned and dimension linear *linearly dimensioned*.

An important structural result for the choice calculus is that any linearly dimensioned expression can be transformed into DCNF, a fact summarized by the following lemmas and theorem.

Any expression e can be transformed into an equivalent expression e' that is maximally choice factored (that is, in CNF). This can be achieved by repeatedly applying the rules from Figure 4 from right to left.

Lemma 1 *$\forall e.\exists e'.e \equiv e' \wedge e'$ is in CNF.*

Lemma 1 is significant on its own, demonstrating that any e can be transformed into CNF, minimizing redundancy. However, only expressions that are linearly dimensioned can, in general, be brought into dimension normal form.

Lemma 2 *If e is linearly dimensioned, then $\exists e'$ in DNF such that $e \equiv e'$.*

From Lemmas 1 and 2 the following result about dimension-choice normal form follows directly.

Theorem 1 *If e is linearly dimensioned, then $\exists e'$ in DCNF such that $e \equiv e'$.*

As an illustration of the three types of normal form, recall the following expressions from Section 3.3.

$$\begin{aligned} e &= \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ 5 + A\langle 1,2\rangle \\ e' &= \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 5+1,5+2\rangle \\ e'' &= 5 + \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle 1,2\rangle \end{aligned}$$

Comparing these to the definitions above, we see that e is in DCNF, while e' is (only) in DNF and e'' is (only) in CNF.

3.5 Variation Design Theory

Not every choice expression is a good variation representation. A trivial example is a choice of the form $A\langle e,e\rangle$ that contains two identical alternatives. Since it does not matter which alternative we select, this is a "false choice" that could be simply replaced by e.

The goal of a variation design theory is to formulate quality criteria for choices and dimensions that can serve as guidelines for the design of variation structures. We briefly describe criteria for identifying spurious choices and dimensions, and transformations to remove them. A more detailed discussion and other design criteria and transformations can be found in [6].

We say that two alternatives e_i and e_j of a choice $D\langle e^n\rangle$ are *equivalent* in context C, written as $e_i \sim_C e_j$, if $[\![C[D\langle e^n\rangle]]\!]$ is unchanged by swapping alternatives e_i and e_j; that is, if

$$[\![C[D\langle e^n\rangle]]\!] = [\![C[D\langle e_1,\ldots,e_j,\ldots,e_i,\ldots,e_n\rangle]]\!]$$

Note that the context in the definition makes it possible to compare expressions that contain free dimensions or variables (whose definition can be given by the context).

We define the function $\bar{\alpha}_{C/i}(e)$ to perform the removal of the ith alternative of a choice in context

C within expression e. This function is defined only if C is a context that matches a choice in e with at least i alternatives; that is, we assume $e = C[D\langle e^n \rangle]$ with $n \geq i$. Then we obtain the following, obvious definition.

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

As an example, consider the following expression.

$$e_{abx} = \mathbf{dim}\ A\langle a,b,x\rangle\ \mathbf{in}\ A\langle 1,1,9\rangle$$

With $C = \mathbf{dim}\ A\langle a,b,x\rangle\ \mathbf{in}\ []$, we can remove the second alternative in the choice in e_{abx} by applying $\bar{\alpha}_{C/2}(e_{abx})$. This yields the expression $\mathbf{dim}\ A\langle a,b,x\rangle\ \mathbf{in}\ A\langle 1,9\rangle$, which is not well dimensioned. This example demonstrates that we cannot, in general, simplify a choice that contains equivalent alternatives in isolation. Since the number of alternatives in a choice must match the number of tags in its binding dimension, reducing the number of alternatives in a choice requires the removal of the corresponding tag in the binding dimension to maintain well dimensionedness. In this example, this would actually work since we have only one choice that is bound by dimension A. But in cases where we have other choices, the removal of the tag is possible only if all corresponding pairs of alternatives in all those other choices are redundant too, which is generally not the case.

On the dimension level, we can consider the equivalence of tags. We define that two tags t_i and t_j are *equivalent* in context C, written as $t_i \sim_C t_j$, if $[\![C[\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e]]\!]$ is unchanged by swapping tags t_i and t_j; that is, if

$$[\![C[\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e]]\!] = [\![C[\mathbf{dim}\ D\langle t_1, \ldots, t_j, \ldots, t_i, \ldots, t_n\rangle\ \mathbf{in}\ e]]\!]$$

Tag equivalence is in a sense a stronger property than equivalence of alternatives since a dimension that defines two equivalent tags can bind many choices, and thus the equivalence has a broader scope. However, equivalent tags do *not* imply equivalent alternatives, which can be seen in the following simple example.

$$e_{ab} = \mathbf{dim}\ A\langle a,b\rangle\ \mathbf{in}\ A\langle A\langle 1,2\rangle, 1\rangle$$

It is clear that selection with either $A.a$ or $A.b$ produces 1 as a result; that is, tags a and b are equivalent. However, neither pair of alternatives in either of the two bound choices is equivalent.

Two equivalent tags are redundant with respect to each other, and therefore, one of them can be safely removed, because the removal is variant preserving. Removing a tag amounts to reducing the size of a dimension and all of its bound choices by one and is done as follows. If $t_i \sim_C t_j$, replace the dimension declaration $\mathbf{dim}\ D\langle t^n\rangle$ with $\mathbf{dim}\ D\langle t_1, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n\rangle$ and every choice $D\langle e^n\rangle$ bound by that dimension with $D\langle e_1, \ldots, e_{j-1}, e_{j+1}, \ldots, e_n\rangle$. We write $\bar{\tau}_{C/t_j}(e)$ for applying this simplification operation to an expression e with $e = C[\mathbf{dim}\ D\langle t^n\rangle\ \mathbf{in}\ e']$. For our examples we obtain the following possible removals.

$$\bar{\tau}_{C/a}(e_{ab}) = \mathbf{dim}\ A\langle b\rangle\ \mathbf{in}\ A\langle 1\rangle \qquad \bar{\tau}_{C/b}(e_{ab}) = \mathbf{dim}\ A\langle a\rangle\ \mathbf{in}\ A\langle A\langle 1\rangle\rangle$$
$$\bar{\tau}_{C/a}(e_{abx}) = \mathbf{dim}\ A\langle b,x\rangle\ \mathbf{in}\ A\langle 1,9\rangle \qquad \bar{\tau}_{C/b}(e_{abx}) = \mathbf{dim}\ A\langle a,x\rangle\ \mathbf{in}\ A\langle 1,9\rangle$$

Since an equivalent tag represents a redundancy in an expression, the systematic removal of equivalent tags does not affect the represented variants. This is summarized in the following theorem.

Theorem 2 $t_i \sim_C t_j \implies \bar{\tau}_{C/t_j}(e) \sim e$

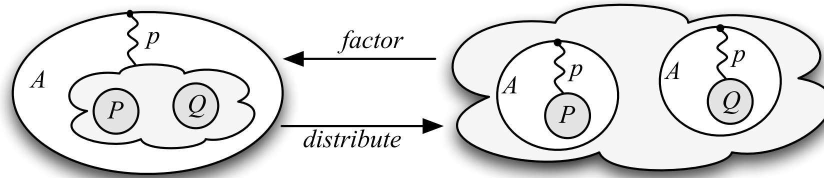

Figure 5: *Factoring in program fields. Factoring amounts to reducing the common context shown for alternatives in choices and thus emphasizes the closeness of different programs within a program field. The formal basis for factoring is provided by the choice commutation rules shown in Figure 4.*

Obviously, Theorem 2 applies to our examples e_{ab} and e_{abx}. In the case of the expression e_{ab}, the tag removal results in a dimension that contains only one tag and corresponding choices with only one alternative each. Such dimensions and choices are trivially superfluous and can thus be eliminated. These and other kinds of transformation are discussed in [6].

3.6 Program Fields

The integrated representation of programs and their variants provided by the choice calculus supports a shift of focus away from single programs to whole sets of closely programs, which we call *program fields* [5]. This is similar to zooming out from a single point in space to a region of surrounding points.

The change of perspective is rather gradual, as illustrated in Figure 5, which shows how two programs $\{A\prec P\succ, A\prec Q\succ\}$ that share a common part (that is, context) A can be represented by factoring out A to produce $A\prec\{P,Q\}\succ$. This process is captured and formalized through the equivalences presented in Section 3.3, specifically the rule C-S from Figure 4. The editing of program fields poses a non-trivial challenge to user-interface design. We have investigated the impact of different concrete representations on the comprehensibility by users. We will present some of the results in Section 3.11.

Program fields can change the process of software maintenance from a discrete, big-step program-to-program hopping approach toward a smoother and more gradual transition between programs via closely related alternatives. Programs are still conceived as discrete points, but they are connected to similar programs in their neighborhood and not just isolated artifacts.

The program-field view also implies a shift from program transformations and program analyses toward program-*field* transformations, which map sets of programs into new sets of programs. When program fields are given by a variation representation, the description of program-field transformations become effectively variation transformations, which must be able to express changes in the context of variation and the addition and removal of variation, see Figure 6.

Similarly to the generalization of program transformations to program-field transformations is the generalization of program analyses to program-*field* analyses. An example for such an analysis is the process of variational type inference, which will be discussed in Sections 3.7 through 3.10.

3.7 Variation Types

The representation of variation in software logically requires compilers and analysis tools to take the variation into consideration and ultimately also produce variational results. One such class of tools are type checkers, which have to produce some of form of variational types to be of general use. As an example, consider two different ways to implement a function in Haskell to find values in a lookup list

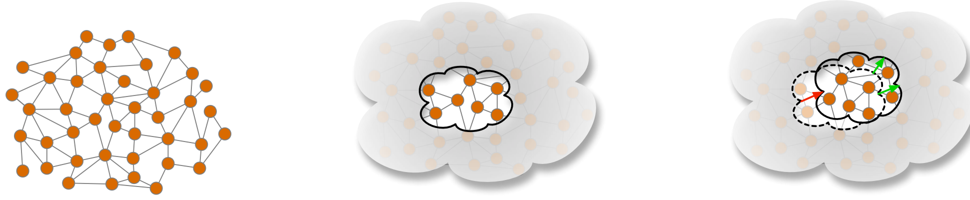

Figure 6: *Visual illustration of the notion of an evolving program field. A program field is a view on a huge set of related program variants. The evolution happens through the addition and removal of variants. The systematic description of such evolutions is an application of variation programming, discussed in Section 3.12.*

of type `[(a,b)]`. In the first, we return a value of type `Maybe b`, possibly containing the first value in the lookup list associated with a given key of type `a`.

```
find x ((k,v):t) | x == k    = Just v
                 | otherwise = find x t
find _ []                    = Nothing
```

In the second, we return a list of type `[b]`, containing all of the values in the lookup list associated with the key.

```
find x ((k,v):t) | x == k    = v : find x t
                 | otherwise = find x t
find _ []                    = []
```

Employing the choice calculus notation introduced in Section 3.1, we can represent the variation between these two function implementations by annotating the program in-place. First, we declare a new dimension of variation, *Res*, representing variation in the function's result. Then we indicate the specific variation points in the code using choices that are bound to the *Res* dimension.

```
dim Res⟨fst,all⟩ in
find x ((k,v):t) | x == k    = Res⟨Just v,v:find x t⟩
                 | otherwise = find x t
find _ []                    = Res⟨Nothing,[]⟩
```

The *Res* dimension declaration above states that we can select one of two tags in the dimension: *fst*, to return the first found value, or *all*, to return all found values. The two choices in the body of the function are synchronized with these tags. For example, if we select the *fst* tag in the *Res* dimension, the first alternative in each of the two choices in the *Res* dimension will also be selected, producing the first function definition above.

The interesting aspect of this example here is that it illustrates that the types inferred in a variational program are, in general, also variational. For our `find` function, we infer the following *variational type* which also contains a choice in the *Res* dimension.[1]

```
find :: a -> [(a,b)] -> Res⟨Maybe b,[b]⟩
```

Variational types, shown in Figure 7, are given by type expressions that are extended by choices. Variational types are like other tree-structured values that can be made variational through the use of the

$$
\begin{array}{rcll}
T & ::= & \tau & \textit{Constant Type} \\
 & | & a & \textit{Type Variable} \\
 & | & T \to T & \textit{Function Type} \\
 & | & D\langle T, ..., T\rangle & \textit{Choice Type} \\
 & | & \bot & \textit{Error Type} \\
 & | & \top & \textit{OK Type}
\end{array}
$$

Figure 7: *Variational types. Alternative types that are not instances of one another can be represented through choices. Variational types are subject to factoring and other equivalence transformations that are valid for the choice calculus in general. Error (and OK) types allow the partial typing of programs in which some of the variants contain type errors.*

---

choice calculus representation.

Constant types, type variables, and function types are as in other type systems—*plain types* contain only these three constructs. Non-plain types may also contain *choice types*. Choice types encode variation in types in the same way that choices encode variation in expressions, with the exception that dimension names in types are globally scoped (see [3] for the rationale of this design decision). The purpose of the error and OK types will be explained later in Section 3.10.

3.8 Variational Type Systems

The purpose of a variational type system is to define the types of programs that contain variations. As illustrated in the Section 3.7 this may lead to choices in the derived type expressions. The key contribution of a variational type system is a set of rules that (1) define the introduction of choice types at the right places and (2) allow the normalization of choice types, which supports a more expressive form of typing through an equivalence predicate on types.

The association of variational types with programs is determined by a set of typing rules, an excerpt of which is shown in Figure 8. The programs that are being typed are expressions of *variational lambda calculus*, which is lambda calculus extended by the constructs of the choice calculus. To keep this presentation simple and focused, we omit the formal definition of variational lambda calculus here as well as much of the technical machinery that is involved in the definition of the type system.

A typing judgment has the form $\Delta, \Gamma \vdash e : T$, which states that expression e has type T in the context of environments Δ and Γ. Environments are implemented as stacks, where $E \oplus (k, v)$ means to push the mapping (k, v) onto environment E, and $E(k) = v$ means that the topmost occurrence of k is mapped to v in E. The Γ environment maps variables to types and is the standard typing environment for lambda calculus. It is used as expected in the typing rules for variables and abstractions. The Δ environment maps expression-level dimension names to globally unique type-level dimension names. These mappings are added by the T-DIM rule and referenced by the T-CHOICE rule. The use of this environment also ensures that every choice is well dimensioned.

Most of the rules are straightforward extensions of the rules found in lambda calculus. The typing of applications is slightly more complex in the presence of variation, however, so the T-APP rule differs from the standard definition. Rather than requiring that the type of the argument and the argument type of the function are equal, we instead require that they are *equivalent* using the equivalence relation $\equiv$ introduced in Section 3.3. The reason for this change is that requiring type equality between the type of

[1] To keep the following discussion simpler, we omit the `Eq` type class constraint on `a`.

$$\frac{c \text{ is a constant of type } \tau}{\Delta,\Gamma \vdash c : \tau}\ \text{T-CON} \qquad \frac{\Delta,\Gamma \oplus (x,T') \vdash e : T}{\Delta,\Gamma \vdash \lambda x.e : T' \to T}\ \text{T-ABS} \qquad \frac{\Gamma(x) = T}{\Delta,\Gamma \vdash x : T}\ \text{T-VAR}$$

$$\frac{\Delta,\Gamma \vdash e_1 : T_1 \qquad \Delta,\Gamma \vdash e_2 : T_2 \qquad T_1 \equiv T_2 \to T}{\Delta,\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP} \qquad \frac{\Delta \oplus (D,D'),\Gamma \vdash e : T \qquad D' \text{ is fresh}}{\Delta,\Gamma \vdash \textbf{dim}\ D\langle t_1,\ldots,t_n\rangle\ \textbf{in}\ e : T}\ \text{T-DIM}$$

$$\frac{\Delta,\Gamma \vdash e_1 : T_1 \qquad \ldots \qquad \Delta,\Gamma \vdash e_n : T_n \qquad \Delta(D) = D'}{\Delta,\Gamma \vdash D\langle e_1,\ldots,e_n\rangle : D'\langle T_1,\ldots,T_n\rangle}\ \text{T-CHOICE}$$

Figure 8: *Typing rules for assigning variational types. Most of the rules are simple extensions of the standard typing rules for lambda calculus. The use of type equivalence in the rule for function application extends the range of type-correct programs.*

---

the argument and the argument type of the function is too strict. We demonstrate this with the following example.

dim $A\langle a,b\rangle$ **in** `succ` $A\langle$`1`,`2`$\rangle$

By the T-DIM typing rule, the type of this expression will be the type of the application in the scope of the dimension. The LHS of the application, `succ`, has type `Int` $\to$ `Int`; the RHS, $A\langle$`1`,`2`$\rangle$, has type $A\langle$`Int`,`Int`$\rangle$. Since `Int` $\neq A\langle$`Int`,`Int`$\rangle$, the T-APP typing rule will fail under a type-equality definition of the $\equiv$ relation. This suggests that equality is too strict a requirement since all of the individual variants generated by the above expression (`succ 1` and `succ 2`) are perfectly well typed (both have type `Int`).

Although the types `Int` and $A\langle$`Int`,`Int`$\rangle$ are not equal, they are still in some sense compatible, and are in fact compatible with a great many other types as well. We can formalize this notion by defining the $\equiv$ type equivalence relation used to determine when function application is well-typed. The example above can be transformed into a more general rule that states that any choice $D\langle T_1,\ldots,T_n\rangle$ is equivalent to type T if all alternative types in the choice type are also equivalent to T. This relationship is captured formally by a choice idempotency rule, which is described in detail with many other other rules in [3].

The most important property of the variational type system is that any plain expression that can be selected from a well-typed variational expression is itself well typed and has a plain type that is obtained by essentially the same selection. This result is formalized in the following theorem, for which we require some auxiliary notation. Since tags are not represented explicitly at the type level, we must extend the notion of tag selection to types using an alternative notion of *selectors*, which are basically indices that identify alternatives in choices by their position. The notation $\overline{D.t}$ ($\overline{s}$) indicates a list of tags (selectors), that is, a decision. The function φ_e is a function derived from e that maps tag sequences to corresponding selector sequences. This is needed since the semantics of variational types use selectors in decisions rather than tags. The function φ_e also manages choice renaming in e since types are *dimension linear* (see Section 3.4).

Theorem 3 (Type preservation) *If* $\varnothing,\Gamma \vdash e : T$ *and* $(\overline{D.t},e') \in [\![e]\!]$, *then* $\Gamma \vdash e' : T'$ *where* $\varphi_e(\overline{D.t}) = \overline{s}$ *and* $(\overline{s},T') \in [\![T]\!]$.

3.9 Variational Unification & Variational Type Inference

The type inference algorithm for variational lambda calculus is an extension of the traditional algorithm $\mathscr{W}$ by Damas and Milner. The most critical part of this extension is an equational unification for variational types that respects the semantics of choice types and allows a less strict typing for function application. The equational theory, called *CT*, is defined by the type equivalence relation sketched in Section 3.8.

To get a sense for the problem of CT-unification, consider the following unification problem.

$$A\langle \texttt{Int}, a\rangle \equiv^? B\langle b, c\rangle$$

Here is a list of potential unifiers for the above problem. In the unifiers, type variables other than a, b, and c are assumed to be fresh.

1. $\sigma_1 = \{a \mapsto \texttt{Int},\ b \mapsto \texttt{Int},\ c \mapsto \texttt{Int}\}$
2. $\sigma_2 = \{b \mapsto A\langle \texttt{Int}, a\rangle,\ c \mapsto A\langle \texttt{Int}, a\rangle\}$
3. $\sigma_3 = \{a \mapsto B\langle \texttt{Int}, f\rangle,\ b \mapsto \texttt{Int},\ c \mapsto A\langle \texttt{Int}, f\rangle\}$
4. $\sigma_4 = \{a \mapsto B\langle f, \texttt{Int}\rangle,\ b \mapsto A\langle \texttt{Int}, f\rangle,\ c \mapsto \texttt{Int}\}$
5. $\sigma_5 = \{a \mapsto B\langle d, f\rangle,\ b \mapsto A\langle \texttt{Int}, d\rangle,\ c \mapsto A\langle \texttt{Int}, f\rangle\}$
6. $\sigma_6 = \{a \mapsto B\langle A\langle i,d\rangle, A\langle j,f\rangle\rangle,\ b \mapsto B\langle A\langle \texttt{Int}, d\rangle, g\rangle, c \mapsto B\langle h, A\langle \texttt{Int}, f\rangle\rangle\}$

After applying any one of these unifiers, the types of the LHS and RHS of the unification problem are equivalent. We observe that σ_6 is the most general of these unifiers. In fact, it is the most general unifier (mgu) for this unification problem; by assigning appropriate types to the type variables in σ_6, we can produce any other unifier. For example, composing σ_6 with $\{i \mapsto d, j \mapsto f, g \mapsto A\langle \texttt{Int}, d\rangle, h \mapsto A\langle \texttt{Int}, f\rangle\}$ yields σ_5, which is in turn the most general among the first five unifiers.

To motivate our approach to unification, consider the following example unification problem.

$$A\langle \texttt{Int}, a\rangle \equiv^? A\langle a, \texttt{Bool}\rangle$$

We might attempt to solve this problem through simple decomposition, by unifying the corresponding alternatives of the choice types. This leads to the unification problem $\{\texttt{Int} \equiv^? a,\ a \equiv^? \texttt{Bool}\}$, which is unsatisfiable. However, notice that $\{a \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}$ is a unifier for the original problem, so this approach to decomposition must be incorrect.

The key insight is that there is a fundamental difference between the type variables in the types a, $A\langle a, T\rangle$, and $A\langle T, a\rangle$, even though all three are named a. A type variable in one alternative of a choice type is *partial* in the sense that it applies only to a subset of the type variants. In particular, it is independent of type variables of the same name in the other alternative of that choice type. In the above example, the two occurrences of a can denote two different types because they cannot be selected at the same time. The important fact that a appears in two different alternatives of the A choice type is lost in the decomposition by alternatives.

We address this problem with a notion of *qualified type variables*, where each type variable is marked by the alternatives in which it is nested. A qualified type variable a is denoted by a_q, where q is the qualification and is given by a set of selectors (see Section 3.8). In addition to the traditional operations of matching and decomposition used in equational unification, our unification algorithm uses two other

operations: choice type *hoisting* and type variable *splitting*. These are needed to transform the types being unified into more similar structures that can then be matched or decomposed; see [3] for details.

We have proved several results about the unification problem, our unification algorithm that uses qualified type variables, and how it correctly implements variational unification. The most important result about variational type inference is that it is sound and complete. Since the algorithm exploits the sharing of common contexts in choices and also the reduction of choices through implicit application of the C-C-MERGE rule, the implementation is much more efficient than the naive, brute-force approach of generating all variants and checking them separately.

All technical details about the correctness of unification and type inference as well as empirical results about the performance of variational type inference can be found in [3].

3.10 Error-Tolerant Variational Type Systems

A limitation of the variational type system described in Section 3.8 is that it is too strict with regard to failure in the sense that it will produce a type error for a whole program field if *any* of the program variants in it contains a type error. In particular, this prevents a user from seeing the types of type-correct invariants, and it also does not provide any information about which variants are type incorrect. To lift these limitations, we have to extend (a) variational types by an error type (as shown in Figure 7) and (b) the type inference rules and algorithm by a mechanism that allows the continuation of type inference in the presence of type errors.

To demonstrate, suppose we add a new dimension of variation to our `find` function introduced in Section 3.7, *Arg*, that captures variation between looking up values based on an example key (as above) or looking up values based on a predicate on keys. We name the tags corresponding to these possibilities *val* and *pred*, respectively.

```
dim Arg⟨val,pred⟩ in
dim Res⟨fst,all⟩ in
find Arg⟨x,p⟩ ((k,v):t)
      | Arg⟨x == k,p k⟩ = Res⟨Just v,v:find x t⟩
      | otherwise       = find Arg⟨x,p⟩ t
find _ []               = Res⟨Nothing,[]⟩
```

Since we can make our selections in the *Res* and *Arg* dimensions independently, this new expression represents four total program variants. We expect variational type inference to infer the following variational type for our new implementation of `find`.

```
find :: Arg⟨a,(a -> Bool)⟩ -> [(a,b)] -> Res⟨Maybe b,[b]⟩
```

But there is an error in the above definition that causes variational type inference to fail. The error is that the variable `x` is unbound in `find x t` if we select *Arg.pred* and *Res.all*.

This can be easily fixed by replacing `x` with the choice *Arg*⟨`x`,`p`⟩. The problem is that the type inference algorithm presented in Section 3.9 provides no hint at the location of this error—it just fails, indicating that there *is* an error.

We have extended variational type inference to return partially correct variational types—that is, variational types containing errors. For example, the errorful variational type of our `find` function can be written as follows, where $\perp$ is a special type that indicates a type error at that location in the type.

```
find :: Arg⟨a,(a -> Bool)⟩ -> [(a,b)] -> Res⟨Maybe b,Arg⟨[b],⊥⟩⟩
```

This type indicates that there is a type error in the result type of the function if the second tag is chosen

from each dimension (*Arg.pred* and *Res.all*). This extension therefore directly supports the location of type errors in variational programs. Similarly, it supports type inference on incomplete variational programs—programs in which only some variants are in a complete and type-correct state—a quality which is needed for incremental development.

The addition of error types is a non-trivial extension to the type system and inference algorithm. In particular, there are many subtle implications for the unification of variational types. In the case of an unbound variable, as above, the location of the error is obvious. However, often there are many possible candidates for the type error, depending on how we infer the surrounding types. The goal is to assign errors such that as few variants as possible are considered ill-typed, that is, to find a type that is *most-defined*. This goal is in addition to the usual goal of inferring the *most general* type possible.

It is not immediately obvious whether these two qualities of types are orthogonal. We have shown that they are and have developed an inference algorithm that identifies most-defined, most-general types.

The essential change to the type system happens in the T-APP rule for typing function applications, extending it to support partial types. Specifically, the premise $T_1 \equiv T_2 \rightarrow T$ is replaced by three conditions that facilitate the introduction of error types in case the equivalence is not achieved.

$$\textsc{T-App}\quad \frac{\Delta,\Gamma \vdash e_1 : T_1 \qquad \Delta,\Gamma \vdash e_2 : T_2 \qquad \uparrow(T_1) = T_2' \rightarrow T' \qquad P = T_2' \bowtie T_2 \qquad T = P \lhd T'}{\Delta,\Gamma \vdash e_1\ e_2 : T}$$

There are essentially two ways that error types can be introduced: (1) If we cannot convert the type of the left argument T_1 into a function type $T_2' \rightarrow T'$, and (2) if T_2' does not match T_2, the type of the argument. The introduction of errors in the second case is handled by matching the two types using a $\bowtie$ operation to produce a typing pattern P, then masking the result type T with P. In the first case, we employ a helper function $\uparrow$, which lifts a function type to the top level, introducing error types as needed.

The details can be found in [2], where we also prove the main results, which are essentially the generalization of the results described in Sections 3.8 and 3.9 to the case of partial/error types.

3.11 Editing Documents Containing Variations

The maintenance of code that contains variation poses several challenges. For example, when editing a piece of variational code it is not always easy to see which parts of different variants go together or depend on one another. Moreover, judging which context information (including definitions of variables, etc.) is effective for which variant can also be a quite difficult task.

The choice calculus has been developed as a formal representation to support working with variations and, as reported above, has proved to be effective in the design and analysis of variational code. One additional important question is how well the choice calculus concepts can support user interfaces for editing variational programs, and how this compares to the use of C Preprocessor (CPP) directives, which are currently the de facto standard for representing variation in code.

To this end, we have developed a GUI representation and implemented a simple prototype for it. An example of the prototype's interface is shown in Figure 9. The prototype is divided into two columns. The left column shows the *dimension area*, where users can choose which program variant they want to see. In the terms of CPP, it is a way of grouping related macros together. The right column of the prototype is called the *code area* and contains the source code of the currently selected program variant. The current selection in the dimension area is called a *configuration*. When users change a configuration on the left, the corresponding code on the right is updated. Code that is highlighted in the code area represents code that is only included for this particular configuration. The colors of the highlighted code

match with the colors of the code's related dimensions. Notice that here we have some blue code inside of the pink code. This is code that is included only if both of the corresponding options are selected in the dimension column.

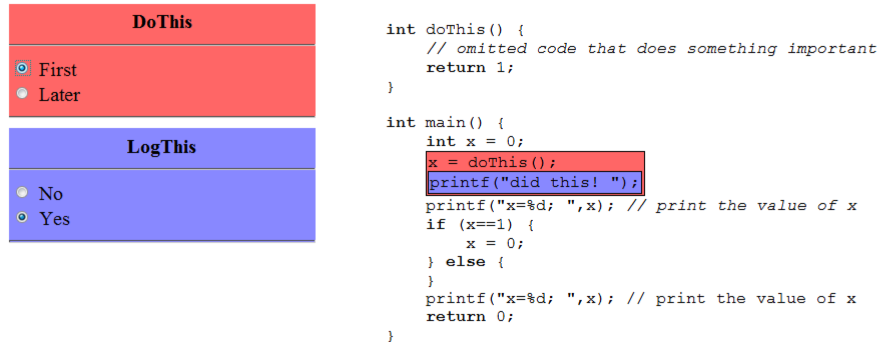

Figure 9: *The variation editor prototype. It provides a visual representation of dimensions and choices. The connection of choices to their binding dimensions is expressed using matching background colors.*

Obviously, our GUI prototype removes noisy CPP syntax and uses colors to highlight code, but in addition to this syntactic aspect, it also provides a virtual form of separation of concerns by only showing the code that is related to the currently selected configuration. However, these features come at a cost: First, there is a need for the user to switch between different configurations. Second, there is a loss of *context* in which a variation point occurs; that is, we can only see the code of one variant at a time, and not how the code differs in other variants. So it is not at all obvious that the proposed representation performs better in terms of software understanding. In fact, there has been prior evidence that including only a subset of these features *does not* significantly improve understanding for some kinds of tasks.

We have performed a user study that has focused primarily on comparing subjects' performance in reading and understanding code in both CPP and the prototype. Specifically, we have tried to assess the accuracy and speed with which participants were able to determine the number of variants in the code and the behavior of specific variants. We recruited 31 undergraduate students as participants that passed a screening test to confirm a basic understanding of C and CPP. This group included 2 female subjects and 29 male subjects. On average, students had taken 4.9 programming courses (3.4 std. dev.) and 1.6 of those involved C or CPP (1.5 std. dev.). 8 of the 31 participants had professional programming experience, and 7 had experience on open-source projects and 2 had both. Of these 13 subjects, 6 used C or CPP at their job or on their open-source projects. 25 out of 31 subjects claimed to use C or CPP in their own personal work.

The results of the study, described in detail in [8], show that participants were significantly more accurate and faster in determining the number of variants using the prototype than using CPP (paired t-test, $t = -15.0721$, $\mathrm{df} = 30$, $p = 1.543 \times 10^{-15}$). Subjects were also significantly more likely to score higher on variant comprehension questions when using the prototype than when using a CPP representation (paired t-test, $t = -4.6032$, $\mathrm{df} = 30$, $p = 7.127 \times 10^{-5}$). These outcomes are matched by the subjects' personal judgments that the prototype was easier to use for the tasks than CPP.

3.12 Variation Programming

The choice calculus offers a *static* representation of variation. The only operation on variational artifacts that it directly supports is that of selection. In practice, variation structures undergo changes like any

other software artifacts. It is important to remember that the choice calculus is itself already representing changes in software by offering choices, but it always does so by providing only a particular snapshot at a specific time. One way to support the maintenance of variation structures that was discussed in Section 3.11 is to offer editing functionality through a graphical user interface. A more general approach is to provide language support to query and transform variation representations.

To explore this idea we have developed a domain-specific embedded language (DSEL) in Haskell for constructing and manipulating variational artifacts. The embedding into Haskell makes it possible to define far-reaching transformations of variation structures while getting certain correctness guarantees through Haskell's expressive type system.

In the DSEL, both the variation representation and any particular object language are represented as data types. The data type for the generic variation representation is given below. It adapts the dimension and choice constructs from the choice calculus into Haskell data constructors, `Dim` and `Chc`. The `Obj` constructor will be explained below. In this definition, the types `Dim` and `Tag` are both synonyms for the predefined Haskell type `String`.

```
data V a = Obj a
         | Dim Dim [Tag] (V a)
         | Chc Dim [V a]
```

The type constructor name `V` is intended to be read as "variational", and the type parameter `a` represents the object language to be varied. So, given a type `Java` representing Java programs, the type `V Java` would represent variational Java programs.

The `Obj` constructor is roughly equivalent to the object structure construct from the choice calculus. However, here we do not explicitly represent the structure as a tree, but rather simply insert an object language value directly. An important feature of the DSEL is that it is possible for the data type representing the object language to itself contain variational types (created by applying the `V` type constructor to its argument types), and operations written in the DSEL can query and manipulate these nested variational values generically. Note that we have omitted the sharing-related constructs from the definition of `V`. This decision is discussed in detail in [7].

To instantiate the variational type to a concrete object language we have to define a data type for that object language. As mentioned it is important that this data type includes references to the `V` type constructor to enable recursively nested occurrences of choices and dimensions. There are different ways of achieving this. The advantages and disadvantages of the different approaches are also discussed in [7].

Finally, having fixed the representation of the choice calculus and the object language, we can define operations to create choices and dimensions, to extend and alter them, or to move them or merge them. This embedding of the choice calculus into Haskell offers a rich environment for exploring all kinds of transformations for maintaining variations. The developed DSEL demonstrates how variation programming tasks can be solved systematically. The scope of the DSEL is limited by the fact that it separates variation representation and transformation into two levels. To address this limitation, we have also developed a generalization of the choice calculus that includes computational features so that choice annotations and transformations can be arbitrarily mixed and nested [9].

4 Personnel Supported

Martin Erwig, Principal Investigator

Eric Walkingshaw, Ph.D. Candidate, graduation expected: 2012

Sheng Chen, Ph.D. Student, graduation expected: 2014

Duc Le, Ph.D. Student, graduation expected: 2014

Aarti Chabra, M.S. awarded 2011

References

[1] S. Chen and M. Erwig. Optimizing the Product Derivation Process. In *IEEE Int. Software Product Line Conference*, pages 35–44, 2011.

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

[3] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *Journal of Functional Programming*, 2012. Under review.

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

[5] M. Erwig and E. Walkingshaw. Program Fields for Continuous Software. In *ACM SIGSOFT Workshop on the Future of Software Engineering Research*, pages 105–108, 2010.

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

[8] 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.

[9] E. Walkingshaw and M. Erwig. A Calculus for Modeling and Implementing Variation. In *ACM SIGPLAN Conf. on Generative Programming and Component Engineering*, 2012. To appear.