A

Extending Type Inference to Variational Programs

SHENG CHEN, MARTIN ERWIG, and ERIC WALKINGSHAW, Oregon State University

Through the use of conditional compilation and related tools, many software projects can be used to generate a huge number of related programs. The problem of typing such variational software is difficult. The brute-force strategy of generating all variants and typing each one individually is (1) usually infeasible for efficiency reasons and (2) produces results that do not map well to the underlying variational program. Recent research has focused mainly on efficiency and addressed only the problem of type checking. In this work we tackle the more general problem of variational type inference and introduce variational types to represent the result of typing a variational program. We introduce the variational lambda calculus (VLC) as a formal foundation for research on typing variational programs. We define a type system for VLC in which VLC expressions are mapped to correspondingly variational types. We show that the type system is correct by proving that the typing of expressions is preserved over the process of variation elimination, which eventually results in a plain lambda calculus expression and its corresponding type. We identify a set of equivalence rules for variational types and prove that the type unification problem modulo these equivalence rules is unitary and decidable; we also present a sound and complete unification algorithm. Based on the unification algorithm, the variational type inference algorithm is an extension of algorithm $\mathcal{W}$. We show that it is sound and complete and computes principal types. We also consider the extension of VLC with sum types, a necessary feature for supporting variational data types, and demonstrate that the previous theoretical results also hold under this extension. Finally, we characterize the complexity of variational type inference and demonstrate the efficiency gains over the brute-force strategy.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*applicative (functional) languages*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Languages, Theory

Additional Key Words and Phrases: variational lambda calculus, variational type inference, variational types

1. INTRODUCTION

The source code in a software project is often used to generate many distinct but related programs that run on different platforms, rely on different libraries, and include different sets of features. Current research on creating and maintaining massively configurable software systems through software product lines [Pohl et al. 2005], generative programming [Czarnecki and Eisenecker 2000], and feature-oriented software development [Apel and Kästner 2009] suggests that the variability of software will only continue to grow. The problem is that even the most basic program verification tools are not equipped to deal with software variation, especially at this scale.

Simple static analyses, such as syntax and type checkers, are defined in terms of single programs, so the usual strategy is to apply these only after generating a particular program variant. But this strategy works only when authors retain control of the source code and clients are provided with a small number of pre-configured, pre-tested programs. If the number of variants to be managed is large or if clients can configure their own customized

Authors' contact: {chensh,erwig,walkiner}@eecs.oregonstate.edu

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

variants, the strategy fails since these errors will be identified only when a particular variant is generated and likely by someone not in a position to either fix or understand them.

Any static analysis defined for single programs can be conceptually extended to variational programs by simply generating all program variants and testing each one individually. In practice, this is usually impossible due to the sheer number of variants that can be generated. The number of variants grows exponentially as new dimensions of variability are added—for example, each independently optional feature multiplies the total number of variants by two—so this brute-force strategy can only be used for the most trivial examples. Sampling techniques can be used to improve the situation, but do not solve the problem in general.

Efficiency is not the only failing of the brute-force strategy; there is also the issue of how to *represent* the results of analyses on variational software. Using the brute-force approach, one type error in the variational program can cause type errors in a huge number of program variants. These errors will be reported in terms of particular variants rather than the actual variational source code. Similarly, the types inferred for a program variant will not correspond to the variational source code, limiting their use as a way to describe and understand the original variational program.

In this paper we address both of these problems in the context of typing. We first present a simple *formal language* to support research on variational static analyses, then define a *type system* for this language where variational programs are assigned correspondingly *variational types*. Finally, we present an algorithm for *variational type inference* that infers variational types from variational programs directly and is therefore significantly more efficient in the expected case than the brute-force strategy.

Other researchers have also addressed the efficiency problem by developing strategies that type check variational software directly, rather than individual program variants [Thaker et al. 2007; Kästner et al. 2012; Kenner et al. 2010]. Our work distinguishes itself primarily by solving the more difficult problem of *type inference*. This leads to some subtle differences. For example, the question of how to represent the results of type inference on variational programs leads to the notion of variational types. This extends the notion of variation from the expression level to the type level, allowing us to represent the types of variational programs more efficiently. More importantly, variational types supports the understanding of variational software by characterizing the variational structure of the encoded types.

In Section 2 we present the *variational lambda calculus* (VLC), a conservative extension of the lambda calculus with constructs for introducing and organizing static variability in lambda calculus expressions. The extension is based on our own previous work on representing variation with the choice calculus [Erwig and Walkingshaw 2011]. VLC provides simple, direct, and structured support for static variability in lambda calculus expressions, enabling arbitrarily fine-grained and widespread variation. We demonstrate this with a few examples emphasizing how variability can impact the types of functions and expressions. In addition to its use in this paper, like the choice calculus, VLC can serve more generally as a formal foundation for theoretical research on variational software.

In Section 3 we develop a type system for VLC that relates variational types to VLC expressions. The structure of variational types is given in Section 3.1, and the typing relation is given by a set of typing rules in Section 3.2. The crucial component of the type system is the rule for function application, which invokes a notion of variational type equivalence. The definition of this equivalence relation is provided in Section 4 together with a corresponding rewriting relationship, normal forms for variational types, and their properties.

In Section 5 we present a precise characterization of the relationship between variational programs, variational types, and the variant programs and types that these represent. VLC expressions are incrementally refined into plain lambda calculus expressions through a process called *selection*. We demonstrate that the typing relation is preserved during selection, eventually yielding plain lambda calculus expressions and their corresponding plain types.

Section 6 presents two independent extensions to the variational type system. Section 6.1 extends the type system with sum types, illustrating that the type system can be extended in a straightforward way to support the additional typing features needed to support real-world functional programming languages, such as Haskell. Section 6.2 describes an extension of the variability metalanguage with a construct for declaring and scoping dimensions of variation. This provides an essential link to our previous work on the choice calculus, showing that VLC is just an instantiation of the choice calculus with the object language of lambda calculus.

The variational type inference algorithm is structurally similar to other type inference algorithms. By far the biggest technical challenge is the unification of variational types. In Section 7 we describe the problem and present a solution in the form of an equational unification algorithm based on a notion of *qualified type variables*. We also evaluate the correctness and analyze the time complexity of this algorithm. In Section 8 we define the type inference algorithm as an extension of algorithm $\mathcal{W}$ and demonstrate that the algorithm is sound, complete, and has the principal type property.

Because a straightforward, brute-force algorithm exists for typing variational programs, as described above, our algorithm must improve on this in order to be practically useful. In Section 9 we characterize the efficiency gains of variational type inference over the brute-force approach, then demonstrate these gains in a series of experiments.

In summary, this paper makes the following contributions.

— The variational lambda calculus language, which can serve as a formal foundation for studying variational typing and other variational analyses (Section 2.2).
— The concept and representation of variational types for characterizing the types of variational expressions (Section 3.1).
— The type system for assigning variational types to VLC expressions (Section 3.2).
— A set of equivalence rules for variational types (Section 4) and a normalizing rewriting relation for checking type equivalence (Theorem 4.6).
— A proof that variational typing is preserved over variant selection (Theorem 5.2). In practical terms, this means that we can determine the type of each plain program variant that can be selected from a variational program, just by applying the same selection to the variational type.
— A proof that variational unification, modulo the type equivalence rules, is unitary (Theorem 7.1).
— An algorithm for solving the variational unification problem (Section 7.2) that is sound (Theorem 7.5) and complete (Theorem 7.6).
— An analysis of the time complexity of the variational unification algorithm (Section 7.4).
— A type inference algorithm for VLC (Section 8) and a proof that the essential properties of type inference for lambda calculus carry over to type inference for VLC, specifically that the algorithm is sound (Theorem 8.1) and complete and most general (Theorem 8.2).
— An evaluation of the efficiency of the type inference algorithm, both analytically and experimentally, that demonstrates the feasibility of our approach (Section 9).

2. VARIATIONAL LAMBDA CALCULUS

After providing a quick overview of the main constructs of our chosen variation representation in Section 2.1, we will define the syntax and semantics of variational lambda calculus (VLC) in Sections 2.2 and 2.3, respectively.

2.1. Elements of the Choice Calculus

Our variation representation is based on our previous work on the *choice calculus* [Erwig and Walkingshaw 2011], a fundamental representation of software variation designed to improve existing variation representations and serve as a foundation for theoretical research in the field. The fundamental concept in the choice calculus is the *choice*, a construct for introduc-

ing variation points in the code. Choices are associated with *dimensions*, which are used to synchronize related choices and provide structure to the variation space.

As an example, suppose we have two different ways to represent some part of a program's state. In one version of the program the state is binary so we use a boolean representation; in another version of the program there are more possible states, so we use integers. The decision of which state to use is *static*—that is, we make the decision before the program runs. In our code, when we initialize this part of the state, we have to somehow choose between these two different representations. This is expressed in the choice calculus as a choice.

```
x = Rep⟨True,1⟩
```

The two different alternatives are tagged with a name, *Rep*, that stands for the decision to be made about the representation. We call *Rep* a *dimension* of variation. Note that in the above example the choice has been chosen minimally, but that is not required. It would have been just as correct to make the whole statement subject to a choice, as shown below.

```
Rep⟨x = True,x = 1⟩
```

Laws for factoring choices and many other transformations are described in our previous work [2011].

Now suppose we have to inspect the value of `x` at some other place in the program. We have to make sure that we process the values with a comparison operator of the right type, as indicated in the example below.

```
if Rep⟨not x,x < 1⟩ then ...
```

This choice ensures that `not` is applied if `x` uses the boolean representation and the numeric comparison is used when `x` is an integer. Here the choice's dimension name comes critically into play. Different choices in different parts of the program will be synchronized if they have the same dimension name. This reflects the fact that we want to make the decision about the state representation once, then reuse this decision consistently throughout our code. In the choice calculus, each such decision is represented by a unique dimension of variation, and each dimension can have many associated choices that will all be synchronized.

So what is the type of `x`? It can be either `Bool` or `Int`, depending on the decision made for the dimension *Rep*. We can therefore express the type of `x` also using a choice.

```
x : Rep⟨Bool,Int⟩
```

Choices can have more than two alternatives, but all choices with the same dimension name must have the same number of alternatives. Of course, a variational program can have choices in many different dimensions, and these can be arbitrarily nested.

The relationship of dimensions and choices, and the kind of variability expressed by the choice calculus can be understood by analogy with conditional compilation using the C Preprocessor (CPP) [GNU Project 2009]. In this analogy, dimensions correspond to CPP macros and choices correspond to `#ifdef` statements that refer to these macros. For example, the above choice might be represented in CPP as shown below.

```
if
  #ifdef REP
    not x
  #else
    x < 1
  #endif
then ...
```

Like CPP, the choice calculus is principally agnostic about the language that is being annotated by choices. However, the choice calculus is a much more structured representation of variability than CPP since:

$$
\begin{array}{rcll}
e & ::= & c & \textit{Constant} \\
& | & x & \textit{Variable} \\
& | & \lambda x.e & \textit{Abstraction} \\
& | & e\ e & \textit{Application} \\
& | & D\langle e,e\rangle & \textit{Choice} \\
& | & \textbf{share}\ v = e\ \textbf{in}\ e & \textit{Binding} \\
& | & v & \textit{Reference}
\end{array}
$$

Fig. 1. VLC syntax.

(1) it annotates the abstract syntax tree, rather than the plain text of the concrete syntax, ensuring that all variants that can be generated are syntactically correct;
(2) choices provide a more restricted and regular form of variability than arbitrary boolean conditions on macros; and
(3) it ensures that variation points are consistent in the sense that all choices in the same dimension must have the same number of alternatives.

Additionally, the choice calculus provides a static sharing construct for factoring out code common to multiple variants, and a construct for declaring and scoping dimensions of variability.

In the following two subsections we introduce the *variational lambda calculus* (VLC). This language is an instance of the more abstract choice calculus, where its simple tree representation of the object language is replaced with constructs of the lambda calculus. Equivalently, VLC can be viewed as a conservative extension of the lambda calculus with the choice and static sharing constructs from the choice calculus. A further extension with the choice calculus's dimension declaration construct is presented in Section 6.2.

2.2. VLC Syntax

The syntax of VLC is given in Figure 1. To simplify the presentation of typing and the equivalence rules, we restrict choices to the binary case. This is not a fundamental limitation since it is easy to simulate an n-ary choice by nesting $n-1$ binary choices.

The first four VLC constructs, *constant*, *variable*, *abstraction*, and *application* are as in other lambda calculi. The *choice*, *binding*, and *reference* constructs are all from the choice calculus. The choice construct was explained in the previous section. The sharing constructs (binding and reference) are similar to let-constructs in lambda calculus, except that they work on the annotation level. They are used to share common subexpressions across different program variants, but are resolved statically like choices. Since **share** is an annotation-level construct that is resolved statically, it does not introduce polymorphism.[1]

Neither VLC nor the choice calculus provide direct support for optional expressions. This is because the alternative model of variation is more general, although in the case of VLC, it can sometimes lead to redundancy. For example, suppose we want to represent a choice between the function `even` or the function `even` applied to the constant `3`. With explicit support for optionality via empty expressions, we might represent this as `even` $A\langle$`3`$,\epsilon\rangle$, where ϵ denotes an empty expression. However, notice that we can't just include ϵ anywhere we allow an expression—for example, the body of an abstraction cannot be empty. Therefore, rather than treat optionality specially, we require the scope of the choice to be expanded. So the above VLC expression would be represented as $A\langle$`even`$,$`even 3`$\rangle$. If the repeated subexpression is large, we could instead factor this out with sharing, for example, **share** $f = e_{big}$ **in** $A\langle f, f$ `3`$\rangle$.

[1]We call the **share** construct **let** in our previous work [Erwig and Walkingshaw 2011] but name it differently here to prevent confusion since it behaves differently than traditional **let** expressions.

$$
\begin{aligned}
\lfloor c \rfloor_s &= c \\
\lfloor x \rfloor_s &= x \\
\lfloor \lambda x.e \rfloor_s &= \lambda x.\lfloor e \rfloor_s \\
\lfloor e_1\ e_2 \rfloor_s &= \lfloor e_1 \rfloor_s\ \lfloor e_2 \rfloor_s \\
\lfloor D\langle e_l, e_r\rangle \rfloor_s &= \begin{cases} \lfloor e_l \rfloor_s & \text{if } s = D \\ \lfloor e_r \rfloor_s & \text{if } s = \tilde{D} \\ D\langle \lfloor e_l \rfloor_s, \lfloor e_r \rfloor_s \rangle & \text{otherwise} \end{cases} \\
\lfloor \textbf{share}\ v = e\ \textbf{in}\ e' \rfloor_s &= \textbf{share}\ v = \lfloor e \rfloor_s\ \textbf{in}\ \lfloor e' \rfloor_s \\
\lfloor v \rfloor_s &= v
\end{aligned}
$$

Fig. 2. Selection / variation elimination.

If all **share**-variable references are bound by corresponding **share** expressions, we say that the expression is *well formed*. If a VLC expression does not contain any choices, bindings, or references (that is, it is a regular lambda calculus expression with constants), we say that the expression is *plain*.

2.3. Semantics

Conceptually, a VLC expression represents a set of related lambda calculus expressions. It is important to stress again that the choice calculus constructs in VLC describe *static* variation in lambda calculus expressions. That is, we will not extend the semantics of lambda calculus to incorporate choices. Rather, the semantics of a VLC expression is a mapping that describes how to produce plain lambda calculus expressions from the VLC expression.

To produce a plain variant, we first expand all **share**-expressions in the usual way. We assume this is done implicitly in the following discussion. Next we must repeatedly eliminate dimensions of variation until we obtain an expression with no choices. For each dimension D, we can select either left or right, which will replace each choice in dimension D with its left or right alternative, respectively. We write $\lfloor e \rfloor_D$ to indicate choosing the left alternatives of all choices in dimension D and $\lfloor e \rfloor_{\tilde{D}}$ to indicate choosing the right alternatives. We call D and $\tilde{D}$ *selectors*, and range over them with the metavariable s. Since each selection eliminates a dimension of variation, we sometimes refer to this operation as *variation elimination*. The operation is defined in Figure 2. Most cases simply propagate the selection to subexpressions. The interesting case is for choices, where the choice is eliminated and replaced by one of its alternatives if the selector is of the corresponding dimension.

We call a set of selectors a *decision*, and range over decisions with the metavariable δ. The semantics of VLC is then defined as a mapping from decisions to plain lambda calculus expressions. The definition is based on repeated selection with selectors taken from decisions, that is, $\lfloor e \rfloor_\delta = \lfloor \lfloor \ldots \lfloor e \rfloor_{s_1} \ldots \rfloor_{s_{n-1}} \rfloor_{s_n}$, where $\delta = \{s_1, s_2, \ldots, s_n\}$. A decision that eliminates all choices in e is called *complete*.

We want the semantics to be "robust" in the sense that selection with a dimension that does not occur in e is well defined but does not eliminate any choice in e. Thus, for any given expression e, a selector in a decision δ can play two different roles when used in a selection. It can either be *relevant* and remove a choice from e or *irrelevant* and not change e. Since there are infinitely many irrelevant dimensions for any expression e, we define the semantics in two conceptual steps. First, we identify the mapping from decisions containing only relevant dimensions, and then extend it to account for irrelevant dimensions. We write $\tilde{s}$ for the inversion of a selector (that is, changing D to $\tilde{D}$ and vice versa) and write $\delta/\tilde{s}$ for exchanging selector s in δ by its inversion $\tilde{s}$. Next we define that a decision δ is *minimally complete* with respect to e if it satisfies the following two conditions.

$$
\begin{array}{rcll}
T & ::= & \tau & \textit{Constant Types} \\
 & | & a & \textit{Type Variables} \\
 & | & T \to T & \textit{Function Types} \\
 & | & D\langle T, T\rangle & \textit{Choice Types}
\end{array}
$$

Fig. 3. VLC types.

(a) $\lfloor e \rfloor_\delta$ is plain, and
(b) $\forall s \in \delta : \lfloor e \rfloor_{\delta - \{s\}}$ is not plain and $\lfloor e \rfloor_\delta \neq \lfloor e \rfloor_{\delta/\bar{s}}$

Condition (a) captures the idea of completeness, and the two conditions in (b) address the minimality constraint: First, if selection without s does not produce a plain expression, s is needed in δ, but only if selection with $\bar{s}$ produces a different result. The latter condition excludes D from a minimally complete decision for an expression $D\langle e, e\rangle$.

The semantics can now be defined by mapping all minimally complete decisions to the plain expressions they select while allowing each decision to also include any irrelevant dimensions. In the definition given below we use the function $|\delta| = \{|s| \mid s \in \delta\}$ (where $|D| = |\tilde{D}| = D$) to extract the set of dimensions underlying a set of selectors.

$$[\![e]\!] = \{(\delta \cup \delta', \lfloor e \rfloor_\delta) \mid \delta \text{ is minimally complete wrt. } e \wedge |\delta| \cap |\delta'| = \varnothing\}$$

To illustrate this definition, we give the semantics of the expression $A\langle e_1, B\langle e_2, e_3\rangle\rangle$ where e_1, e_2, and e_3 are plain expressions, and where we use the shorthand notation $\{\delta : e\}$ to denote the set $\{(\delta \cup \delta', e) \mid |\delta| \not\subseteq |\delta'|\}$.

$$[\![A\langle e_1, B\langle e_2, e_3\rangle\rangle]\!] = \{\{A\} : e_1\} \cup \{\{\tilde{A}, B\} : e_2\} \cup \{\{\tilde{A}, \tilde{B}\} : e_3\}$$

Note that due to the second part of condition (b) dimension B does not necessarily appear in the first entry since it is irrelevant. Note also that it is because of this condition that the following equality holds.

$$[\![B\langle A\langle e_1, e_2\rangle, A\langle e_1, e_3\rangle\rangle]\!] = [\![A\langle e_1, B\langle e_2, e_3\rangle\rangle]\!]$$

The semantics is the basis for the equivalence rules of choice expressions that we will make use of extensively in the rest of this paper.

3. TYPE SYSTEM

In this section we present a type system for VLC. We present a representation of variational types in Section 3.1, and typing rules for relating variational types to VLC expressions in Section 3.2.

The type system is based on the definition of an equivalence relation on variational types. In order to test for type equivalence, we also have to identify a representative instance from each equivalence class. This is achieved through a set of terminating and confluent rewrite rules. This technical aspect is provided in Section 4. Finally, in Section 5 we present one of our main results, which says that typing is preserved over selection.

3.1. Variational Types

As the example in Section 2.1 demonstrates, describing the type of variational programs requires a similar notion of variational types. The representation of variational types for VLC is given in Figure 3. The meanings of constant types, type variables, and function types are similar to other type systems. We extend the notion of plainness to types, defining that *plain types* contain only these three constructs. Non-plain types contain *choice types* to represent variation in types, just as choices represent variation in expressions. Choice types often (but do not always) correspond directly to choice expressions; for example, the expression $D\langle 2, \texttt{True}\rangle$ has the corresponding choice type $D\langle \texttt{Int}, \texttt{Bool}\rangle$.

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

$$\text{T-APP}\ \frac{\Lambda, \Gamma \vdash e : T'' \qquad \Lambda, \Gamma \vdash e' : T' \qquad T'' \equiv T' \to T}{\Lambda, \Gamma \vdash e\ e' : T}$$

$$\text{T-SHARE}\ \frac{\Lambda, \Gamma \vdash e' : T' \qquad \Lambda \oplus (v, T'), \Gamma \vdash e : T}{\Lambda, \Gamma \vdash \textbf{share}\ v = e'\ \textbf{in}\ e : T} \qquad \text{T-REF}\ \frac{\Lambda(v) = T}{\Lambda, \Gamma \vdash v : T}$$

$$\text{T-CHOICE}\ \frac{\Lambda, \Gamma \vdash e_1 : T_1 \qquad \Lambda, \Gamma \vdash e_2 : T_2}{\Lambda, \Gamma \vdash D\langle e_1, e_2 \rangle : D'\langle T_1, T_2 \rangle}$$

Fig. 4. VLC typing rules.

In the next section we define the mapping from VLC expressions to variational types.

3.2. Typing Rules

The association of types with expressions is determined by a set of typing rules, given in Figure 4. These rules also ensure that all **share**-bound variable references are bound by corresponding **share** expressions.

We use two separate environments in the typing rules, Λ and Γ, each implemented as a stack. The notation $E \oplus (k, v)$ means to push the mapping (k, v) onto the environment stack E, and the notation $E(k) = v$ means that the topmost occurrence of k is mapped to v in E. The Λ environment maps **share**-bound variables to the type of their bound expression. The Γ environment is the standard typing environment for lambda calculus variables. The use of separate environments for lambda calculus variables and **share**-bound variables is a presentation decision to emphasize that these variables inhabit different namespaces and serve different roles. However, they could be easily merged into a single environment.

The T-CON rule is a trivial rule for mapping constant expressions to type constants. The T-ABS and T-VAR typing rules for typing abstractions and variables use the typing environment Γ and are the same as in other type systems for simply-typed lambda calculi without explicit type annotations [Pierce 2002].

The typing of applications is more complicated 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 be equal, we instead require that the left expression be *equivalent* to a function of the appropriate type, using the type equivalence relation $\equiv$. We will return to the T-APP rule in Section 4, where we show why type equivalence is needed and define the type equivalence relation.

The T-SHARE and T-REF rules describe the typing of **share** expressions and their corresponding variable references. These rules use the Λ environment, as described above, and are otherwise straightforward.

The T-CHOICE rule states that the type of a choice is a choice type in the same dimension, where each alternative in the choice type is the type of the corresponding alternative in the choice expression.

$$\textsc{Fun}\ \frac{T'_l \equiv T'_r \qquad T_l \equiv T_r}{T'_l \to T_l \equiv T'_r \to T_r} \qquad \text{F-C}\ \ D\langle T_1, T_2\rangle \to D\langle T'_1, T'_2\rangle \equiv D\langle T_1 \to T'_1, T_2 \to T'_2\rangle$$

$$\text{C-C-Swap1}\ \ D'\langle D\langle T_1, T_2\rangle, T_3\rangle \equiv D\langle D'\langle T_1, T_3\rangle, D'\langle T_2, T_3\rangle\rangle \qquad \text{C-C-Swap2}\ \ D'\langle T_1, D\langle T_2, T_3\rangle\rangle \equiv D\langle D'\langle T_1, T_2\rangle, D'\langle T_1, T_3\rangle\rangle$$

$$\text{C-C-Merge1}\ \ D\langle D\langle T_1, T_2\rangle, T_3\rangle \equiv D\langle T_1, T_3\rangle \qquad \text{C-C-Merge2}\ \ D\langle T_1, D\langle T_2, T_3\rangle\rangle \equiv D\langle T_1, T_3\rangle \qquad \textsc{Choice}\ \frac{T_1 \equiv T'_1 \qquad T_2 \equiv T'_2}{D\langle T_1, T_2\rangle \equiv D\langle T'_1, T'_2\rangle}$$

$$\text{C-Idemp}\ \frac{T_1 \equiv T \qquad T_2 \equiv T}{D\langle T_1, T_2\rangle \equiv T} \qquad \textsc{Refl}\ \ T \equiv T \qquad \textsc{Symm}\ \frac{T \equiv T'}{T' \equiv T} \qquad \textsc{Trans}\ \frac{T \equiv T' \qquad T' \equiv T''}{T \equiv T''}$$

Fig. 5. Variational type equivalence.

4. TYPE EQUIVALENCE

In this section we return to the discussion of the T-App rule from Section 3.2. This rule is similar to the standard rule for typing application in lambda calculus, except that requiring type equality between the type of the argument and the argument type of the function is too strict. We demonstrate this with the following example.

```
succ A⟨1,2⟩
```

The LHS of the application, `succ`, has type `Int → Int`; the RHS, $A\langle 1,2\rangle$, has type $A\langle \texttt{Int}, \texttt{Int}\rangle$. Since $\texttt{Int} \neq A\langle \texttt{Int}, \texttt{Int}\rangle$, 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 $A\langle \texttt{Int}, \texttt{Int}\rangle$ are not equal, they are still in some sense compatible, and are in fact compatible with an infinite number of other types as well. In this section we 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 type $D\langle T_1, T_2\rangle$ is equivalent to type T if both alternative types T_1 and T_2 are also equivalent to T. This relationship is captured formally by the choice idempotency rule, C-Idemp, one of several type equivalence rules given in Figure 5.

Besides idempotency, there are many other type equivalence rules concerning choice types. The F-C rule states that we can factor/distribute function types and choice types. The C-C-Swap rules state that types that differ only in the nesting of their choice types are equivalent. The C-C-Merge rules reveal the property that outer choices *dominate* inner choices. For example, $D\langle D\langle 1,2\rangle, 3\rangle$ is semantically equivalent to $D\langle 1,3\rangle$ since the selection of the first alternative in the outer choice implies the selection of the first alternative in the inner choice.

The remaining rules are very straightforward. The Fun rule propagates equivalency across function types, defining that two function types are equivalent if their argument and result types are equivalent. Similarly, the Choice equivalence rule propagates equivalency across choice types, defining that two choice types are equivalent if both of their alternatives are equivalent. The Refl, Symm, and Trans rules make type equivalence reflexive, symmetric, and transitive, respectively.

In our previous work we provide a set of semantics-preserving transformation laws for choice calculus expressions [2011]. The type equivalence relation is directly descended from these laws.

The important property of equivalent types is that they represent the same mapping from *super-complete* decisions to plain types. A super-complete decision on types T_1 and T_2 is a decision that is complete for both T_1 and T_2; that is, it resolves both potentially variational types into plain types. Making a selection on types is the same as making a selection on expressions. As with expressions, we define selection in irrelevant dimensions to be idempotent, for example, $\lfloor \texttt{Int} \rfloor_s = \texttt{Int}$. This is crucial since super-complete decisions will often result in selection in dimensions that do not exist in one type or the other. The semantics function $[\![\cdot]\!]$ for types is defined in the same way as for expressions, as a mapping from all complete decisions to the plain types they produce through the process of selection.

The following lemma states that a super-complete decision on two equivalent types produces the same plain type.

LEMMA 4.1 (TYPE EQUIVALENCE PROPERTY). $T_1 \equiv T_2 \implies \forall \delta \in S : (\delta, T_1') \in [\![T_1]\!] \wedge (\delta, T_2') \in [\![T_2]\!] \implies T_1' = T_2'$, *where S is the set of all super-complete decisions on T_1 and T_2.*

The proof of Lemma 4.1 relies on a simpler lemma that states that type equivalency is preserved over selection.

LEMMA 4.2 (TYPE EQUIVALENCE PRESERVATION). *If $T_1 \equiv T_2$, then $\lfloor T_1 \rfloor_s \equiv \lfloor T_2 \rfloor_s$.*

A proof sketch for this lemma is given in the appendix, in Section A.3.1. Using Lemma 4.2, we can now prove Lemma 4.1.

PROOF OF LEMMA 4.1. By induction over the size of δ, from Lemma 4.2 it follows that $T_1 \equiv T_2$ implies $\lfloor T_1 \rfloor_\delta \equiv \lfloor T_2 \rfloor_\delta$. Since δ is complete for both types, then neither $\lfloor T_1 \rfloor_\delta$ or $\lfloor T_2 \rfloor_\delta$ will have any choice types. By examining the four equivalence rules that do not include choice types (FUN, REFL, SYMM, TRANS), it is clear that these types must be structurally identical. □

A simple observation is that each equivalence class of types is infinite. For example, we can always trivially expand a type T into an equivalent choice type $D\langle T, T\rangle$. In order to facilitate checking whether two types are equivalent, we identify one representative from each equivalence class as a normal form and define rewriting rules to achieve this normal form. Two types are then equivalent if they have the same normal form.

We define the *type simplification* relation $\rightsquigarrow$ to rewrite a type into a simpler one and use the reflexive, transitive closure of this relation $\rightsquigarrow^*$ to transform a type into normal form. The type simplification rules, shown in Figure 6, are derived from the type equivalence rules in Figure 5.

The FUN equivalence rule leads to two rewriting rules, S-F-ARG and S-F-RES, which simplify the argument and result of a function type, respectively. Likewise, the S-F-C-ARG and S-F-C-RES rules are derived from the F-C equivalence rule, and factor a choice type out of the argument or result of a function type, respectively.

The two S-C-SWAP simplification rules are derived from the two C-C-SWAP equivalence rules, but each adds an additional premise that ensures choice types will be nested according to a known ordering relation $\preceq$ on dimension names. Thus, if T is in normal form and $A \preceq B$, then no choice type $A\langle \ldots \rangle$ will appear within an alternative of a choice type $B\langle \ldots \rangle$ in T.

Picking a good ordering relation is important for efficiency since the S-C-SWAP rules each duplicate part of the type. In the worst case, the normalization process could lead to an exponential blow up in the size of the type. For example, if we assume a lexicographic ordering for $\preceq$, then normalizing the type $D\langle C\langle B\langle A\langle \cdots \rangle, \cdots \rangle, \cdots \rangle, T\rangle$, will lead to T being duplicated eight times. One way to deal with this problem is to determine $\preceq$ heuristically, for example, by initially performing a depth-first traversal of one of the types we are comparing (before converting it into normal form), then using its nesting of choices as $\preceq$. This will ensure that we only perform the necessary choice-swaps in the other type. Another approach (which can be combined with the heuristic approach) is to share, rather than duplicate, the common part when swapping choice types. This requires solving the common subexpression problem,

$$\text{S-F-Arg}\ \frac{T_l \rightsquigarrow T_l'}{T_l \to T_r \rightsquigarrow T_l' \to T_r} \qquad \text{S-F-Res}\ \frac{T_r \rightsquigarrow T_r'}{T_l \to T_r \rightsquigarrow T_l \to T_r'}$$

$$\text{S-F-C-Arg}\ D\langle T_1, T_2\rangle \to T \rightsquigarrow D\langle T_1 \to T, T_2 \to T\rangle \qquad \text{S-F-C-Res}\ T \to D\langle T_1, T_2\rangle \rightsquigarrow D\langle T \to T_1, T \to T_2\rangle$$

$$\text{S-C-Swap1}\ \frac{D \preceq D'}{D'\langle D\langle T_1, T_2\rangle, T_3\rangle \rightsquigarrow D\langle D'\langle T_1, T_3\rangle, D'\langle T_2, T_3\rangle\rangle}$$

$$\text{S-C-Swap2}\ \frac{D \preceq D'}{D'\langle T_1, D\langle T_2, T_3\rangle\rangle \rightsquigarrow D\langle D'\langle T_1, T_2\rangle, D'\langle T_1, T_3\rangle\rangle}$$

$$\text{S-C-Dom1}\ \frac{\lfloor T_1 \rfloor_D = T_1' \qquad T_1 \neq T_1'}{D\langle T_1, T_2\rangle \rightsquigarrow D\langle T_1', T_2\rangle} \qquad \text{S-C-Dom2}\ \frac{\lfloor T_2 \rfloor_{\bar{D}} = T_2' \qquad T_2 \neq T_2'}{D\langle T_1, T_2\rangle \rightsquigarrow D\langle T_1, T_2'\rangle}$$

$$\text{S-C-Alt1}\ \frac{T_1 \rightsquigarrow T_1'}{D\langle T_1, T_2\rangle \rightsquigarrow D\langle T_1', T_2\rangle} \qquad \text{S-C-Alt2}\ \frac{T_2 \rightsquigarrow T_2'}{D\langle T_1, T_2\rangle \rightsquigarrow D\langle T_1, T_2'\rangle} \qquad \text{S-C-Idemp}\ D\langle T, T\rangle \rightsquigarrow T$$

Fig. 6. Variational type simplification.

for which there are some efficient algorithms [Downey et al. 1980; Nelson and Oppen 1980]. Since performance is not our primary focus here, our unification algorithm uses a variant of the first approach, only swapping choices as needed during the decomposition process.

The remaining rewriting rules are straightforward. Like the S-F-Arg and S-F-Res rules, the S-C-Alt rules are focused adaptations of the Choice equivalence rule, one simplifies the left alternative of a choice type, the other simplifies the right. The S-C-Dom rules follow less directly from the corresponding C-C-Merge equivalence rules. They reuse the selection operation from the semantics to more immediately eliminate any dominated choice types. Finally, the S-C-Idemp rewriting rule is derived from the C-Idemp equivalence rule, but is somewhat stricter since it requires the two alternatives to be structurally equal.

By repeatedly applying type simplification until no rewrite rule matches, we achieve a type in normal form. Types in normal form satisfy the following criteria:

1. Choice types are maximally lifted over function types. For example, the type $A\langle \texttt{Int} \to a, a \to \texttt{Bool}\rangle$ is in normal form, while $A\langle \texttt{Int}, a\rangle \to A\langle a, \texttt{Bool}\rangle$ is not.
2. The type does not contain dominated choices. For example, the type $A\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, a\rangle$ is not in normal form. It can be simplified to $A\langle \texttt{Int}, a\rangle$, which is in normal form.
3. The nesting of choices adheres to the ordering relation $\preceq$ on their dimension names.
4. The type contains no choice types with equivalent alternatives.
5. Finally, a function type is in normal form if both its argument and result types are in normal form; a choice type is in normal form if all its alternatives are in normal form.

Figure 7 shows an example transformation of the type $B\langle A\langle \tau_1, \tau_2\rangle, A\langle \tau_1, \tau_2\rangle\rangle$ into its corresponding normal form $A\langle \tau_1, \tau_2\rangle$ (we assume $A \preceq B$). Note that in the application of the S-C-Swap1 rule, we arbitrarily chose to swap the nested choice in the first alternative. We

$$
\begin{aligned}
& B\langle A\langle \tau_1, \tau_2\rangle, A\langle \tau_1, \tau_2\rangle\rangle \\
& \rightsquigarrow A\langle B\langle \tau_1, A\langle \tau_1, \tau_2\rangle\rangle, B\langle \tau_2, A\langle \tau_1, \tau_2\rangle\rangle\rangle && \text{(S-C-SWAP1)} \\
& \rightsquigarrow A\langle B\langle \tau_1, \tau_1\rangle, B\langle \tau_2, \tau_2\rangle\rangle && \text{(S-C-DOM)} \\
& \rightsquigarrow A\langle \tau_1, B\langle \tau_2, \tau_2\rangle\rangle && \text{(S-C-IDEMP/S-C-ALT1)} \\
& \rightsquigarrow A\langle \tau_1, \tau_2\rangle && \text{(S-C-IDEMP/S-C-ALT2)}
\end{aligned}
$$

Fig. 7. Example transformation into normal form.

could have also applied S-C-SWAP2, or applied S-C-IDEMP to the alternatives of the A choice type. An important property of the $\rightsquigarrow^*$ relation, however, is that our decisions at these points do not matter. No matter which rule we apply, we will still achieve the same normal form. This is the property of *confluence*, expressed in Theorem 4.3 below.

THEOREM 4.3 (CONFLUENCE). *If $T \rightsquigarrow^* T_1$ and $T \rightsquigarrow^* T_2$, then there exists a T' such that $T_1 \rightsquigarrow^* T'$ and $T_2 \rightsquigarrow^* T'$.*

A rewriting relation is confluent if it is both *locally confluent* and *terminating*. These properties are expressed for the $\rightsquigarrow^*$ relation below, in Lemmas 4.4 and 4.5. From these two lemmas, Thorem 4.3 follows directly.

LEMMA 4.4 (LOCAL CONFLUENCE). *For any type T, if $T \rightsquigarrow T_1$ and $T \rightsquigarrow T_2$, then there exists some type T' such that $T_1 \rightsquigarrow^* T'$ and $T_2 \rightsquigarrow^* T'$.*

The proof of this lemma is given in the appendix, in Section A.3.2.

LEMMA 4.5 (TERMINATION). *Given any type T, $T \rightsquigarrow^* T'$ is terminating.*

We give an informal proof of termination here to convince the reader. A formal proof based on a counting mechanism is presented in the appendix, in Section A.3.3.

PROOF SKETCH. The $\rightsquigarrow^*$ relation will terminate when we reach a normal form (as defined by the criteria listed above) because an expression satisfying these criteria will not match any rule in the $\rightsquigarrow$ relation, by construction. Therefore, we must show that these criteria will be satisfied in a finite number of steps.

Trivially, the two S-C-DOM rules eliminate dominated choices, and the S-C-IDEMP rule eliminates equivalent alternatives, in a finite number of steps. The S-F-C-RES and S-F-C-ARG lift choice types over function types, and no rule can lift function types back out. The S-C-SWAP1 and S-C-SWAP2 rules define a similarly one-way relation for choice nestings, according to the $\preceq$ relation on choice names. Thus, we can see that all rules make progress toward satisfying one of the criteria, and that, in isolation they can achieve this in a finite number of steps.

A potential challenge to termination arises via the duplication of type subexpressions in the S-F-C and S-C-SWAP rules. For example, the right alternative T_3 of the original choice type is duplicated in the application of the S-C-SWAP1 rule. However, observe that these can only create a finite amount of additional work since the rules otherwise make progress as described above. □

A terminating rewriting relation is by definition *normalizing*. Since rewriting is both confluent and normalizing, any variational type can be transformed into a unique normal form [Baader and Nipkow 1998, p. 12]. We write $norm(T)$ for the unique normal form of T. We capture the fact that a normal form represents an equivalence class by stating in the following theorem that two types are equivalent if and only if they have the same normal form.

THEOREM 4.6. $T \equiv T' \Leftrightarrow norm(T) = norm(T')$.

PROOF. This follows from Theorem 4.3, the fact that $\rightsquigarrow^*$ is normalizing, and the observation that the $\equiv$ relation is the symmetric, reflexive, and transitive closure of $\rightsquigarrow$. □

This is the essential result needed for checking type equivalence.

5. TYPE PRESERVATION

An important property of the type system is that any plain expression that can be selected from a well-typed variational expression is itself well typed, and that the plain type of the variant can be obtained by the same selection on the variational type. This result can be proved with the help of the following lemma, which states that variational typing is preserved over a single selection.

LEMMA 5.1. $\Lambda, \Gamma \vdash e : T \implies e$ *is plain or* $\forall s\colon \Lambda, \Gamma \vdash \lfloor e \rfloor_s : \lfloor T \rfloor_s$.

PROOF. The proof is based on induction over the typing rules. We show only the cases for the T-APP rule and the T-CHOICE rule. The cases for the other rules can be constructed similarly. Also, we write the typing judgment $\Lambda, \Gamma \vdash e : T$ more succinctly as $e : T$ when the environments are not significant.

We consider the T-APP rule first. Assume that $e\ e' : T$, then we must show that $\lfloor e\ e' \rfloor_s : \lfloor T \rfloor_s$. We do this through the following sequence of observations.

1.	$e : T''$, $e' : T'$, and $T'' \equiv T' \to T$	by the definition of T-APP
2.	$\lfloor e\ e' \rfloor_s = \lfloor e \rfloor_s\ \lfloor e' \rfloor_s$	by the definition of $\lfloor \cdot \rfloor_s$
3.	$\lfloor e \rfloor_s : \lfloor T'' \rfloor_s$ and $\lfloor e' \rfloor_s : \lfloor T' \rfloor_s$	by the induction hypothesis
4.	$\lfloor T'' \rfloor_s \equiv \lfloor T' \to T \rfloor_s$	by 1 and Lemma 4.2
5.	$\lfloor T'' \rfloor_s \equiv \lfloor T' \rfloor_s \to \lfloor T \rfloor_s$	by 4 and the definition of $\lfloor \cdot \rfloor_s$
6.	Therefore, $\lfloor e\ e' \rfloor_s : \lfloor T \rfloor_s$	by 2, 3, 5, and the definition of T-APP

For the T-CHOICE rule, assume that $D\langle e_1, e_2 \rangle : D\langle T_1, T_2 \rangle$. Then we must show that $\lfloor D\langle e_1, e_2 \rangle \rfloor_s : \lfloor D\langle T_1, T_2 \rangle \rfloor_s$. There are two cases to consider: either s represents a selection in choice D, or it does not.

If s represents a selection in choice D, the proof follows directly from the induction hypothesis and the definitions of selection on expressions and types. For example, if s selects the first alternative in D, then selecting the first alternative on both sides of the typing relation leaves us with $\lfloor e_1 \rfloor_s : \lfloor T_1 \rfloor_s$, which is the induction hypothesis.

If s does not represent a selection in D, then applying selection to each side of the typing relation yields $D\langle \lfloor e_1 \rfloor_s, \lfloor e_2 \rfloor_s \rangle : D\langle \lfloor T_1 \rfloor_s, \lfloor T_2 \rfloor_s \rangle$. Since $\lfloor e_1 \rfloor_s : \lfloor T_1 \rfloor_s$ and $\lfloor e_2 \rfloor_s : \lfloor T_2 \rfloor_s$ by the induction hypothesis, the claim follows through a direct application of the T-CHOICE rule. □

By induction it follows that for any set of selectors δ that yields a plain expression from e, δ also selects the corresponding plain type from T. Therefore, the following theorem, which captures the type preservation property described at the beginning of this section, follows directly from Lemma 5.1.

THEOREM 5.2 (TYPE PRESERVATION). *If* $\varnothing, \Gamma \vdash e : T$ *and* $(\delta, e') \in [\![e]\!]$, *then* $\Gamma \vdash e' : T'$ *where* $(\delta, T') \in [\![T]\!]$.

With Theorem 5.2, the type of any particular variant of e can be easily selected from its inferred variational type T. For example, suppose $\varnothing, \Gamma \vdash e : T$ with $T = A\langle B\langle T_1, T_2 \rangle, T_3 \rangle$, then the type of $e' = \lfloor \lfloor e \rfloor_A \rfloor_{\bar{B}}$ is $\lfloor \lfloor T \rfloor_A \rfloor_{\bar{B}} = T_2$.

Type preservation demonstrates that the type system is correct. We must also show that it is complete. That is, if every plain variant encoded by a variational expression e is type correct, then our type system will assign a variational type to e. The completeness property is stated in the following theorem.

THEOREM 5.3 (COMPLETENESS). *If* $\forall(\delta, e') \in [\![e]\!]$, $\exists T'$ *such that* $\varnothing, \Gamma \vdash e' : T'$, *then* $\exists T$ *such that* $\varnothing, \Gamma \vdash e : T$.

This theorem can be proved by simple induction over the structure of e, with the help of the following lemma.

LEMMA 5.4. *If* D *is a dimension used in a choice in* e, *and* $\exists T_1$ *such that* $\Lambda, \Gamma \vdash \lfloor e \rfloor_D : T_1$, *and* $\exists T_2$ *such that* $\Lambda, \Gamma \vdash \lfloor e \rfloor_{\tilde{D}} : T_2$, *then* $\exists T$ *such that* $\Lambda, \Gamma \vdash e : T$.

PROOF. This follows from induction over the structure of e and the typing derivations of $\lfloor e \rfloor_D$ and $\lfloor e \rfloor_{\tilde{D}}$. □

6. EXTENSIONS

This section considers two different ways of extending VLC and its type system (two dimensions of variability, if you will). Section 6.1 discusses the interaction of choice types and other traditional typing features, such as sum types. Section 6.2 extends the variation metalanguage of VLC with dimension declarations.

6.1. Type System Extensions

To extend the VLC type system from lambda calculus to a full-fledged functional language such as Haskell we need to add more features. In this section we briefly outline the necessary steps with a few examples to illustrate how to deal with the interaction of variational types with standard features of type systems. Specifically, we consider how to add sum types, which are the basis for data types in functional languages.

Adding a new typing feature requires at least the extension of VLC's expression syntax, variational types, and the typing rules. In the case of sum types we add expressions **inl** e and **inr** e as well as a **case** expression for pattern matching expressions built using **inl** e and **inr** e [Pierce 2002]. We add the type $T_1 + T_2$ to denote a sum type, and we need to add typing rules for all new syntactic forms. The rules for **inl** and **inr** are exactly the same as presented by Pierce [2002], except for the additional environment used in the typing judgment. The typing rule for **case**, which is slightly different, is shown below.

T-CASE
$$\frac{\Lambda, \Gamma \vdash e_0 : T_1 + T_2 \qquad \Lambda, \Gamma \oplus (x_1, T_1) \vdash e_1 : T_1' \qquad \Lambda, \Gamma \oplus (x_2, T_2) \vdash e_2 : T_2' \qquad T_1' \equiv T_2'}{\Lambda, \Gamma \vdash \textbf{case}\ e_0\ \textbf{of inl}\ x_1 \Rightarrow e_1 \mid \textbf{inr}\ x_2 \Rightarrow e_2 : T_1'}$$

Much like in the case of function application, the branches of a case statement do not need to have the same type. Instead, we only require that their types be equivalent.

The next step is the extension of the type equivalence relation and type simplification rules. For sum types, we get two new equivalence rules. The rule SUM states that two sum types are equivalent if their corresponding left and right types are equivalent. Also, sum types are distributive over choice types, as shown by the rule S-C.

SUM
$$\frac{T_1 \equiv T_1' \qquad T_2 \equiv T_2'}{T_1 + T_2 \equiv T_1' + T_2'}$$

S-C
$$D\langle T_1 + T_1', T_2 + T_2' \rangle \equiv D\langle T_1, T_2 \rangle + D\langle T_1', T_2' \rangle$$

The new simplification rules can be straightforwardly derived from the equivalence rules.

S-S-L
$$\frac{T_l \rightsquigarrow T_l'}{T_l + T_r \rightsquigarrow T_l' + T_r}$$

S-S-R
$$\frac{T_r \rightsquigarrow T_r'}{T_l + T_r \rightsquigarrow T_l + T_r'}$$

S-S-C
$$D\langle T_1, T_2 \rangle + D\langle T_1', T_2' \rangle \rightsquigarrow D\langle T_1 + T_1', T_2 + T_2' \rangle$$

Extensions with tuple types, recursive types, and parametric types follow the same pattern and require the extension of expressions, types, typing rules, equivalence rules, and rewriting

rules, as discussed above. All these extensions are rather straightforward and don't present any problems as far as the interaction with choice types is concerned.

6.2. Local Dimension Declarations

A distinguishing feature of the choice calculus, not present in VLC, is that the dimension names associated with choices are locally bound and scoped. This is achieved through an additional *dimension declaration* construct. A dimension declaration $\mathbf{dim}\, D\langle t,u\rangle\, \mathbf{in}\, e$ introduces a dimension named D, which can be used by choices only within the expression e (called the dimension's *scope*). The names t and u are called *tags* and give names to the left and right alternatives that we can select in dimension D. So we can refer to the selector D with the dimension-qualified tag $D.t$, and the selector $\tilde{D}$ with the dimension-qualified tag $D.u$. Each dimension declaration corresponds to a single decision that must be made in order to resolve a variational expression into a plain one. This means that if there are multiple declarations of the dimension D, each of these dimensions of variability will be independent.

Local dimension declarations promote modularity since they allow subexpressions to be composed without the risk of choices becoming unintentionally synchronized. For example, in a language without local dimension declarations, suppose a library defines a function f that varies in dimension A. A client of the library, unaware of its implementation, applies f to an argument e that also varies in a dimension A. When the user applies f to e, these two dimensions will be unintentionally synchronized. This can be observed in the semantics of $f\ e$, which has only two entries, one for selecting the first alternative in A, the other for selecting the second.

$$f = A\langle f_1, f_2\rangle$$
$$e = A\langle e_1, e_2\rangle$$
$$[\![f\ e]\!] = \{(A, f_1\ e_1),\ (\tilde{A}, f_2\ e_2)\}$$

By making the scope of dimensions explicit, local dimension declarations avoid this problem. The semantics of $f\ e$ below has four alternatives since the decisions about the two A dimensions are independent.

$$f = \mathbf{dim}\, A\langle t,u\rangle\, \mathbf{in}\, A\langle f_1, f_2\rangle$$
$$e = \mathbf{dim}\, A\langle x,y\rangle\, \mathbf{in}\, A\langle e_1, e_2\rangle$$
$$[\![f\ e]\!] = \{([A.t, A.x], f_1\ e_1),\ ([A.t, A.y], f_1\ e_2),\ ([A.u, A.x], f_2\ e_1),\ ([A.u, A.y], f_2\ e_2)\}$$

Note that in the presence of dimension declarations, we represent decisions by a list of qualified tags.[2]

Local dimension name scoping is an obviously desirable quality, but it turns out to pose several challenges in the context of typing. In order to provide continuity with our previous work and clarify the relationship of VLC to the choice calculus, we outline those challenges here. Then we describe how the type system can be extended to support local dimension declarations. Since this extension adds complexity that is incidental to the core issue of variational typing, we confine this extension to this subsection and assume global dimension names throughout the rest of the paper.

We can extend the syntax of VLC expression to support local dimension declarations by just adding the **dim** construct to the existing syntax, as shown below.

$$\begin{array}{rcll} e & ::= & \ldots & \textit{(VLC syntax as before)} \\ & | & \mathbf{dim}\, D\langle t,t\rangle\, \mathbf{in}\, e & \textit{Dimension declaration} \end{array}$$

[2]There are at least two ways to cope with the ambiguity of independent dimensions with overlapping tag names. One is to disallow it, enforcing that tag names in like-named dimensions must be distinct. The other, which we use in our previous work, is to enforce that tags must be selected from dimensions in a fixed order [2011].

We also add the constraint that every choice $D\langle e_1, e_2\rangle$ be within the scope of a corresponding dimension declaration $\mathbf{dim}\, D\langle t, u\rangle\, \mathbf{in}\, e$.

Initially, it seems we can just make a corresponding extension at the type level, adding dimension declarations to types, just as we extended them with choices.

$$\begin{array}{rcll} T & ::= & \ldots & \textit{(VLC types as before)} \\ & | & \mathbf{dim}\, D\langle t, t\rangle\, \mathbf{in}\, T & \textit{Type-level dimension declaration} \end{array}$$

We will enforce that all choices are bound by dimension declarations directly in the typing rules. To do this, we introduce a new environment Δ. The new T-DIM rule extends Δ with the name of the declared dimension, and the revised T-CHOICE rule enforces the constraint by referring to Δ. Just as with choices, the type of a dimension declaration is a corresponding dimension declaration at the type level. The following typing rules are marked with stars since we will show later that they are not correct.

T-DIM*

$$\frac{\Delta \oplus D, \Lambda, \Gamma \vdash e : T}{\Delta, \Lambda, \Gamma \vdash \mathbf{dim}\, D\langle t_1, t_2\rangle\, \mathbf{in}\, e : \mathbf{dim}\, D\langle t_1, t_2\rangle\, \mathbf{in}\, T}$$

T-CHOICE*

$$\frac{\Delta, \Lambda, \Gamma \vdash e_1 : T_1 \qquad \Delta, \Lambda, \Gamma \vdash e_2 : T_2 \qquad D \in \Delta}{\Delta, \Lambda, \Gamma \vdash D\langle e_1, e_2\rangle : D\langle T_1, T_2\rangle}$$

One problem we can observe with this extension is that typing can result in *dimension duplication* at the type level. Consider the following simple example, where $c_1 : \tau_1$ and $c_2 : \tau_2$.

$$(\lambda x.\lambda f.f\ x\ x)\ (\mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, A\langle c_1, c_2\rangle)$$

This example represents two variants. If we select $A.t$, we get the expression $(\lambda x.\lambda f.f\ x\ x)\ c_1$. If we select $A.u$, we get $(\lambda x.\lambda f.f\ x\ x)\ c_2$. However, the extended type system identifies the following variational type with *four* type variants!

$$(\mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, A\langle \tau_1, \tau_2\rangle \to \mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, A\langle \tau_1, \tau_2\rangle \to a) \to a$$

The problem is that after the dimension type is added to the typing environment Γ by the T-ABS rule, it is retrieved and inserted by the T-VAR rule twice, once each time the variable x is referenced.

The proper type for the above expression, in which the dimension type is not duplicated, can be achieved by factoring the dimension declaration out of the type as shown below.

$$\mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, (A\langle \tau_1, \tau_2\rangle \to A\langle \tau_1, \tau_2\rangle \to a) \to a$$

Unfortunately, this solution does not work in general. This is demonstrated by the next example, which declares two independent dimensions named A (we underline one of these dimension names for illustrative purposes only), and which cannot be correctly typed using type-level dimension declarations without resorting to dimension renaming.

$$(\lambda x.\mathbf{dim}\, \underline{A}\langle r, s\rangle\, \mathbf{in}\, \lambda f.f\ \underline{A}\langle x, c_1\rangle\ x)\ \mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, A\langle c_1, c_2\rangle$$

If we naively apply our extended typing rules to this example, we get the following type.

$$\begin{array}{l} \mathbf{dim}\, \underline{A}\langle r, s\rangle\, \mathbf{in} \\ \quad (\underline{A}\langle \mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, A\langle \tau_1, \tau_2\rangle, \tau_2\rangle \to \mathbf{dim}\, A\langle t, u\rangle\, \mathbf{in}\, A\langle \tau_1, \tau_2\rangle \to a) \to a \end{array}$$

Here we have again duplicated the dimension declaration in the argument, which declares dimension A. This time, however, we cannot factor out the repeated dimension type because doing so would *capture* the outermost choice in dimension $\underline{A}$. Since the scope of a dimension in an expression is not preserved through the process of typing, there is no way to represent the desired type with locally scoped dimensions. The best we can do is a type that is equivalent up to dimension renaming, such as the following.

$$\begin{array}{l} \mathbf{dim}\, A\langle r, s\rangle\, \mathbf{in}\, \mathbf{dim}\, A'\langle t, u\rangle\, \mathbf{in} \\ \quad (A\langle A'\langle \tau_1, \tau_2\rangle, \tau_2\rangle \to A'\langle \tau_1, \tau_2\rangle \to a) \to a \end{array}$$

This fact significantly weakens any argument in favor of locally-scoped dimensions in types. Additionally, the constraint of avoiding dimension duplication in the typing rules is extremely arduous. Typing naturally involves a great deal of type duplication—types are duplicated whenever variables are referenced multiple times and frequently during the unification of type variables (see Section 7). Furthermore, the simple factoring transformation shown above often fails in the presence of dependent dimensions and dimensions of the same name, forcing us to resort to dimension renaming for many kinds of expressions.

To solve these problems, our extension to support locally scoped dimensions in expressions still uses globally scoped dimension names in types. This means that we extend the language of VLC with dimension declarations, as shown above, but we *do not* extend the language of VLC types. The correct typing rules for this extension are given below. Compared to the discarded starred typing rules, T-DIM* and T-CHOICE*, we do not introduce dimension declarations into types and we have some extra machinery to support the mapping of expression-level dimensions names to the corresponding names at the type-level.

$$\text{T-DIM}\quad \frac{\Delta \oplus (D, D'), \Lambda, \Gamma \vdash e : T \qquad D' \text{ is fresh}}{\Delta, \Lambda, \Gamma \vdash \mathbf{dim}\, D \langle t_1, t_2 \rangle \,\mathbf{in}\, e : T} \qquad \text{T-CHOICE'}\quad \frac{\Delta, \Lambda, \Gamma \vdash e_1 : T_1 \qquad \Delta, \Lambda, \Gamma \vdash e_2 : T_2 \qquad \Delta(D) = D'}{\Delta, \Lambda, \Gamma \vdash D\langle e_1, e_2 \rangle : D'\langle T_1, T_2 \rangle}$$

The T-DIM rule generates a fresh dimension name D' for the newly declared dimension D and stores a mapping from D to D' in Δ. This is used by the T-CHOICE' rule to produce a corresponding choice type with the appropriate type-level dimension name. The resulting type of a dimension declaration is just the type of its scope.

This provides a simple solution to the problem of typing variation in a metalanguage with locally scoped dimensions. This is relevant both because of VLC's relationship to the choice calculus and because local dimension scoping has other desirable qualities unrelated to typing. The main advantage of this solution is that the extension is modular in the sense that it does not require any significant changes to the rest of the type system. We must only extend the other typing rules to propagate the dimension environment along. Therefore, we do not consider local dimension declarations further.

7. UNIFYING VARIATIONAL TYPES

The type inference algorithm for VLC is an extension of the traditional algorithm $\mathcal{W}$ by Damas and Milner [1982]. The extension consists mostly of 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 is defined by the type equivalence relation in Figure 5. We call this unification problem *choice type* (CT).

The properties of the CT-unification problem are described in Section 7.1, while the unification algorithm that solves it is presented in Section 7.2. In Section 7.3 we formally evaluate the correctness of the unification algorithm, and we analyze its time complexity in Section 7.4. Once this groundwork has been laid, the variational type inference algorithm itself is straightforward. It is is given in Section 8.

7.1. The Choice Type Unification Problem

If we view a choice as a binary operator on its two subexpressions, then CT's equational theory contains both distributivity (introduced by the C-C-SWAP rule) and associativity (which follows from the C-C-MERGE rules). Usually, this yields a unification problem that is undecidable [Anantharaman et al. 2004]. CT-unification, however, *is* decidable. The key insight is that a normalized choice type cannot contain nested choice types in the same dimension, effectively bounding the number of choice types a variational type can contain.

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

$$A\langle \texttt{Int}, a \rangle \stackrel{?}{\equiv} B\langle b, c \rangle \tag{1}$$

Several potential unifiers for this problem are given below. In each mapping, type variables other than a, b, and c are assumed to be fresh.

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

These mappings are unifiers since, after applying any one of these mappings to the types in problem (1), the types of the LHS and RHS of the 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 CT-unification problem. This means that 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.

Although σ_6 is more general than σ_5, if we apply either one to the types in problem (1), then simplify dominated choices, we will get the same result. Therefore, it may seem that the generality provided by σ_6 is superficial. But in fact, σ_6 solves strictly more unification problems than σ_5. For instance, assume $e_1 : A\langle \texttt{Int}, a\rangle \to c$, $e_2 : B\langle b, c\rangle$, and $e_3 : B\langle \texttt{Bool}, \texttt{Int}\rangle$. Using σ_5 the expression $e_1\ e_2$ has type $A\langle \texttt{Int}, f\rangle$, so the expression $e_1\ e_2\ e_3$ will be ill-typed since $A\langle \texttt{Int}, f\rangle$ does not unify with $B\langle \texttt{Bool}, \texttt{Int}\rangle$. On the other hand, if we use σ_6, then $e_1\ e_2$ has type $B\langle h, A\langle \texttt{Int}, e_1\rangle\rangle$, so the expression $e_1\ e_2\ e_3$ is type correct since the unification problem $B\langle h, A\langle \texttt{Int}, f\rangle\rangle \equiv^? B\langle \texttt{Bool}, \texttt{Int}\rangle$ has the mgu $\{f \mapsto B\langle l, A\langle m, \texttt{Int}\rangle\rangle, h \mapsto B\langle \texttt{Bool}, k\rangle\}$, where k, l and m are fresh type variables.

An equational unification problem is said to be *unitary* if there is a unique unifier that is more general than all other unifiers [Baader and Snyder 2001]. This is important to make type inference feasible since we need only maintain the unique mgu throughout the inference process.

It is not immediately obvious that CT-unification is unitary. Usually, equational unification problems with associativity and distributivity are not unitary. However, the same bounds that make CT-unification decidable (that is, the normalization process ensures that there are no nested choices in the same dimension, via the S-C-DOM rules) also make the problem unitary. Specifically, choice dominance ensures that a CT-unification problem can be decomposed into a finite number of simpler unification problems that are known to be unitary. Furthermore, the mgus of these subproblems can be used to construct the unique mgu of the original CT-unification problem.

That the CT-unification problem is unitary is captured in the following theorem.

THEOREM 7.1. *Given a CT-unification problem U, there is a unifier σ such that for any unifier σ', there exists a mapping θ such that $\sigma' = \theta \circ \sigma$.*

The proof of this theorem relies on definitions from the rest of this section and so is delayed until the appendix, in Section A.1. We give a high-level description of the argument here.

A CT-unification problem encodes a finite number of plain subproblems, where a plain unification problem is between two plain types. For example, problem (1) encodes the plain subproblems $\texttt{Int} \equiv^? b$, $\texttt{Int} \equiv^? c$, $a \equiv^? b$, and $a \equiv^? c$. In principle, solving a variational unification problem requires solving all of the plain unification problems it encodes. One challenge of CT-unification is that different plain subproblems may share the same type variables, and these may be incorrectly mapped to different types when solving the different subproblems. However, CT-unification problems whose plain subproblems share no common type variables

are easy to solve. We just generate all of the plain subproblems, solve each of them using the traditional Robinson unification algorithm [Robinson 1965], then take the union of the resulting set of unifiers as the solution to the original problem.

The basic structure of the argument that CT-unification is unitary is therefore to demonstrate that:

1. We can transform any CT-unification problem U into an equivalent unification problem U', such that the plain subproblems of U' share no type variables. This can be done through the process of type variable qualification, described in Section 7.2.
2. These subproblems are plain and therefore themselves unitary.
3. We can construct a unique mgu for U from the mgus of the individual subproblems of U'. This is achieved through the process of completion, also described in Section 7.2.

Of course, we do not actually solve CT-unification problems by solving all of the corresponding plain subproblems separately since it would be very inefficient. Type variable qualification and completion all do play a role in the actual algorithm, however, which is developed and presented in the next subsection.

7.2. Qualified Type Unification

This section will present our approach to unifying variational types. Since there is no general algorithm or strategy for equational unification problems [Baader and Snyder 2001], we begin by motivating our approach. Consider the following example unification problem.

$$A\langle \texttt{Int}, a\rangle \stackrel{?}{\equiv} A\langle a, \texttt{Bool}\rangle \tag{2}$$

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 $\{\texttt{Int} \stackrel{?}{\equiv} a,\ a \stackrel{?}{\equiv} \texttt{Bool}\}$, which is unsatisfiable. However, notice that $\{a \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}$ is a unifier for the original problem (through choice domination), 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 example (2), 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 4), rendered as a lexicographically sorted sequence. For example, the type variable a in $B\langle T_1, A\langle a, T_2\rangle\rangle$ corresponds to the qualified type variable $a_{A\bar{B}}$. Likewise, the (non-qualified) unification problem in example (1) can be transformed into the *qualified unification problem* $A\langle \texttt{Int}, a_{\bar{A}}\rangle \stackrel{?}{\equiv}_q B\langle b_B, c_{\bar{B}}\rangle$, and the problem in example (2) can be transformed into $A\langle \texttt{Int}, a_{\bar{A}}\rangle \stackrel{?}{\equiv}_q A\langle a_A, \texttt{Bool}\rangle$.

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.

$$\begin{array}{r} A\langle \texttt{Int}, \underline{a_{\tilde{A}}} \rangle \equiv_q^? B\langle b_B, c_{\tilde{B}} \rangle \\ \scriptstyle split\downarrow \\ \underline{A\langle \texttt{Int}, B\langle a_{\tilde{A}B}, a_{\tilde{A}\tilde{B}} \rangle \rangle} \equiv_q^? B\langle b_B, c_{\tilde{B}} \rangle \\ \scriptstyle hoist\downarrow \\ B\langle A\langle \texttt{Int}, a_{\tilde{A}B} \rangle, A\langle \texttt{Int}, a_{\tilde{A}\tilde{B}} \rangle \rangle \equiv_q^? B\langle b_B, c_{\tilde{B}} \rangle \end{array}$$

$$\begin{array}{r} A\langle \texttt{Int}, a_{\tilde{A}B} \rangle \equiv_q^? b_B \\ A\langle \texttt{Int}, a_{\tilde{A}\tilde{B}} \rangle \equiv_q^? c_{\tilde{B}} \end{array}$$

Fig. 8. Example of qualified unification.

Hoisting is applied when unifying two types that have top-level choice types with different dimension names. To illustrate, consider the following unification problem.

$$A\langle B\langle \texttt{Int}, \texttt{Bool} \rangle, a_{\tilde{A}} \rangle \equiv_q^? B\langle a_B, \texttt{Bool} \rangle$$

We cannot immediately decompose this problem by alternatives since the dimensions of the top-level choice types do not match. However, this problem can be solved by applying the C-C-SWAP1 rule to the LHS, thereby hoisting the B choice type to the top.

$$B\langle A\langle \texttt{Int}, a_{\tilde{A}B} \rangle, A\langle \texttt{Bool}, a_{\tilde{A}\tilde{B}} \rangle \rangle \equiv_q^? B\langle a_B, \texttt{Bool} \rangle$$

Notice that we must add a qualification to all of the duplicated type variables that were originally in the alternative opposite the hoisted choice type, but are now nested beneath it, such as the $a_{\tilde{A}}$ variable in the example. Now we can decompose the problem by unifying the corresponding alternatives of the top-level choice type.

Splitting is the expansion of a type variable into a choice type between two qualified versions of that variable. It is used whenever decomposition cannot proceed and the problem cannot be solved by hoisting. For example, to decompose the problem $a \equiv_q^? A\langle a_A, \texttt{Int} \rangle$, we first split a into the choice type $A\langle a_A, a_{\tilde{A}} \rangle$, then decompose by alternatives. To decompose the problem $A\langle \texttt{Int}, a_{\tilde{A}} \rangle \equiv_q^? B\langle \texttt{Int}, b_{\tilde{B}} \rangle$, we can split either $a_{\tilde{A}}$ into a choice in B or $b_{\tilde{B}}$ into a choice in A. In either case, we must then apply hoisting once before the problem can be decomposed by alternatives.

Figure 8 presents an example in which split and hoist are used to prepare a qualified unification problem for decomposition. Note that after decomposition, we do not need to split b_B into a choice in A because b_B is isolated and occurs on only one side of the subtask; instead we can return the substitution $\{b_B \mapsto A\langle \texttt{Int}, a_{\tilde{A}B} \rangle\}$ for this subtask directly. Likewise for $c_{\tilde{B}}$ in the second subtask.

To solve a unification problem U, we solve the corresponding qualified unification problem Q, then transform the solution of Q, σ_Q, into a solution for U, σ_U. Each mapping $a \mapsto T$ in σ_U is derived through a process called *completion* from the related subset of mappings in σ_Q, $\{a_{q_1} \mapsto T_1, \ldots, a_{q_n} \mapsto T_n\}$. Each qualified mapping $a_{q_i} \mapsto T_i$ describes a leaf in a tree of nested choice types that makes up the resulting type T. Building and populating this tree is the goal of completion. For example, given the qualified mappings $\{a_A \mapsto \texttt{Int}, a_{\tilde{A}B} \mapsto b, a_{\tilde{A}\tilde{B}} \mapsto \texttt{Bool}\}$, completion yields the unqualified mapping $a \mapsto A\langle \texttt{Int}, B\langle b, \texttt{Bool} \rangle \rangle$. When the qualified mappings do not describe a complete tree, the completion process introduces fresh type variables to represent the unconstrained parts of the type. For example, given the qualified mappings $\{a_A \mapsto \texttt{Int}, a_{\tilde{A}\tilde{B}} \mapsto \texttt{Bool}\}$, completion yields the unqualified mapping $a \mapsto A\langle \texttt{Int}, B\langle c, \texttt{Bool} \rangle \rangle$, where c is a fresh type variable.

Formally, we define completion by folding the helper function *comp*, defined in Figure 9, across the mappings in σ_Q. This function produces a partially completed type given (1) a type variable qualification q, (2) the type to store at the path described by q, and (3) the type that is being completed. The definition of *comp* relies on top-down pattern matching on the first

$$
\begin{aligned}
comp(Dq, T, D\langle T_1, T_2\rangle) &= D\langle comp(q, T, T_1), T_2\rangle \\
comp(\tilde{D}q, T, D\langle T_1, T_2\rangle) &= D\langle T_1, comp(q, T, T_2)\rangle \\
comp(Dq, T, T') &= D\langle comp(q, T, T'), fresh(T')\rangle \\
comp(\tilde{D}q, T, T') &= D\langle fresh(T'), comp(q, T, T')\rangle \\
comp(\epsilon, T, a) &= T
\end{aligned}
$$

Fig. 9. Helper function used in the completion process.

and third arguments (ϵ matches the empty qualification), and on a second helper function *fresh* that renames every type variable in its argument type to a new, fresh type variable.

In the first two cases of *comp*, if the partially completed type already contains a choice type in the dimension D referred to by the first selector in the qualification, the function consumes the selector and propagates the completion into the appropriate alternative. Note that these choice types will have been created by a previous invocation of *comp* on a different qualification, as we'll see below. In the third and fourth cases, the partially completed type does not already contain a choice type in D, so we create a new one and propagate the completion into the appropriate branch, freshening the type variables in the duplicated alternative. In these first four cases, we traverse and create a tree structure of choice types. This relies on the fact that selectors are sorted in the qualification q, avoiding the creation of choice types in the same dimension. However, it is possible that types stored at the leaves of this tree will contain choice types in dimensions created by *comp*; these can be eliminated by a subsequent normalization step.

Finally, the completion of $a \mapsto T$ from $\{a_{q_1} \mapsto T_1, \ldots, a_{q_n} \mapsto T_n\}$ is defined as follows.

$$T = comp(q_1, T_1, comp(q_2, T_2, \ldots\ comp(q_n, T_n, b)\ \ldots))$$

The initial argument to the folded completion function is a fresh type variable b, and the order in which we process the qualifications $q_1, \ldots, q_n$ does not matter. Also note that, although *comp* is not specified for all argument patterns, the completion process cannot fail on any unifier produced by our unification algorithm. This is because we do not produce mappings for "overlapping" qualified type variables (see the discussion of *occurs* later in this section).

The final and most important piece of the variational-type-unification puzzle is the algorithm for solving qualified unification problems. The definition of this algorithm, *unify*, is given in Figure 10. In this definition, we use p to range over plain types (which do not contain choice types), and g to range over *ground plain types*, which do not contain choice types or type variables. We also assume that $D_1 \neq D_2$ and use T_L and T_R to refer to the entire LHS and RHS of the unification problem, respectively. Cases marked with an asterisk represent two symmetric cases. That is, the definition of $unify^*(T, T')$ implies the definition of both $unify(T, T')$, as written, and a dual case $unify(T', T) = unify(T, T')$.

The definition of *unify* will be explained in detail below. The algorithm relies on several helper functions. The function *hoist* implements a deep form of the C-C-SWAP rule. It takes as arguments a choice type T and a dimension name D of a (possibly nested) choice type in T, returning a type equivalent to T but with a D choice type at the root. For example, $hoist(A\langle B\langle a, \texttt{Int}\rangle, \texttt{Bool}\rangle, B)$ yields $B\langle A\langle a, \texttt{Bool}\rangle, A\langle \texttt{Int}, \texttt{Bool}\rangle\rangle$. The function *choices* takes a type and returns the set of dimension names of all choice types it contains. The function *splittable* returns the type variables that can be split into a choice type. A variable is splittable if the path from itself to the root consists only of choice types; that is, there are no function types between the root of the type and the type variable. For example, $splittable(A\langle \texttt{Int} \to b, c\rangle) = \{c\}$. The final helper function, *occurs*, performs an operation similar to an occurs check, described in the final case below.

Finally, we can describe each case of the *unify* algorithm as follows.

$$unify : (T_L, T_R) \to \sigma$$

$$unify(p, p') = robinson(p, p') \tag{1}$$

$$unify^*(a_q, D\langle T_1, T_2\rangle) = unify(D\langle a_{Dq}, a_{\tilde{D}q}\rangle, D\langle T_1, T_2\rangle) \tag{2}$$

$$unify(D\langle T_1, T_2\rangle, D\langle T_1', T_2'\rangle) = \sigma_1 \leftarrow unify(T_1, T_1') \quad \sigma_2 \leftarrow unify(T_2\sigma_1, T_2'\sigma_1) \quad \text{return } \sigma_1 \circ \sigma_2 \tag{3}$$

$$unify^*(D_1\langle T_1, T_2\rangle, D_2\langle T_1', T_2'\rangle) \mid D_2 \in choices(T_L) = unify(hoist(T_L, D_2), T_R) \tag{4}$$

$$unify^*(D_1\langle T_1, T_2\rangle, D_2\langle T_1', T_2'\rangle) \mid splittable(T_L) \neq \varnothing \wedge D_2 \notin choices(T_L) = a_q \leftarrow splittable(T_L) \quad \theta \leftarrow \{a_q \mapsto D_2\langle a_{D_2q}, a_{\tilde{D}_2q}\rangle\} \quad \text{return } unify(hoist(T_L\theta, D_2), T_R) \tag{5}$$

$$unify(D_1\langle T_1, T_2\rangle, D_2\langle T_1', T_2'\rangle) \mid splittable(T_L) = \varnothing \wedge D_2 \notin choices(T_L) \wedge splittable(T_R) = \varnothing \wedge D_1 \notin choices(T_R) = unify(D_2\langle T_L, T_L\rangle, T_R) \tag{6}$$

$$unify^*(g, D\langle T_1, T_2\rangle) = \sigma_1 \leftarrow unify(g, T_1) \quad \text{return } \sigma_1 \circ unify(g, T_2\sigma_1) \tag{7}$$

$$unify^*(T \to T', D\langle T_1, T_2\rangle) = unify(D\langle T \to T', T \to T'\rangle, D\langle T_1, T_2\rangle) \tag{8}$$

$$unify(T_1 \to T_2, T_1' \to T_2') = \sigma \leftarrow unify(T_1, T_1') \quad \text{return } \sigma \circ unify(T_2\sigma, T_2'\sigma) \tag{9}$$

$$unify^*(a_q, T \to T') \mid occurs(a_q, T_R) = \text{fail} \mid \text{otherwise} = \{a_q \mapsto T_R\} \tag{10}$$

Fig. 10. The qualified unification algorithm.

(1) When unifying two plain types, we defer to Robinson's unification algorithm [1965].

(2) To unify a type variable with a choice type, we split the type variable as described earlier in this section.

(3) To unify two choice types in the same dimension, we decompose the problem and unify their corresponding alternatives.

(4) To unify two choice types in different dimensions, we try to hoist a choice type so that both types are rooted by a choice in the same dimension.

(5) If this is impossible, then a splittable type variable is split into a choice in that dimension, which can then be hoisted.

(6) To unify two choice types in different dimensions, where there is no splittable type variable, we partially decompose the problem by unifying each alternative of one choice type with the other choice type.

(7) To unify a ground plain type g with a choice type, we again decompose the problem, unifying g with each alternative of the choice type.

(8) To unify a function type with a choice type in dimension D, we first expand the function type into a choice type in D, similar to the splitting operation on type variables. We then decompose the problem by alternatives.

(9) To unify two function types, we unify their corresponding argument types and return types, composing the results.

$$D_1\langle \underset{split\downarrow}{a_q}, D_3\langle T_1, b_r\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$
$$D_1\langle \underset{hoist\downarrow}{D_2}\langle a_{D_2q}, a_{\tilde{D}_2q}\rangle, D_3\langle T_1, b_r\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$
$$D_2\langle D_1\langle a_{D_2q}, D_3\langle T_1, b_{D_2r}\rangle\rangle, D_1\langle a_{\tilde{D}_2q}, D_3\langle T_1, b_{\tilde{D}_2r}\rangle\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$

$$D_1\langle a_q, D_3\langle T_1, \underset{split\downarrow}{b_r}\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$
$$D_1\langle a_q, D_3\langle T_1, \underset{hoist\downarrow}{D_2}\langle b_{D_2r}, b_{\tilde{D}_2r}\rangle\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$
$$D_1\langle a_q, \underset{hoist\downarrow}{D_2}\langle D_3\langle T_1, b_{D_2r}\rangle, D_3\langle T_1, b_{\tilde{D}_2r}\rangle\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$
$$D_2\langle D_1\langle a_{D_2q}, D_3\langle T_1, b_{D_2r}\rangle\rangle, D_1\langle a_{\tilde{D}_2q}, D_3\langle T_1, b_{\tilde{D}_2r}\rangle\rangle\rangle \equiv_q^? D_2\langle T_2, T_3\rangle$$

Fig. 11. Demonstration of variable independence.

(10) Finally, to unify a type variable with a function type, a process similar to an *occurs check* is needed. The operation $occurs(a_q, T)$ returns true if there exists a type variable $a_{q'}$ in T such that $q \subseteq q'$. This ensures that we do not assign overlapping type variables to different types, supporting the completion of a qualified unifier back into an unqualified unifier.

7.3. Correctness of the Unification Algorithm

In this subsection we collect several results to demonstrate the correctness of the qualified unification algorithm.

We begin by observing that the operations of decomposition, splitting, and hoisting form the core of the algorithm. In the following lemmas we establish the correctness of these operations. First, we show that the decomposition by alternatives of a qualified unification problem is correct.

LEMMA 7.2 (DECOMPOSITION). *Let $T_L = D\langle T_1, T_2\rangle$ and $T_R = D\langle T'_1, T'_2\rangle$. Then $T_L \equiv_q^? T_R$ is unifiable iff $T_1 \equiv_q^? T'_1$ and $T_2 \equiv_q^? T'_2$ are unifiable. Moreover, if the problem is unifiable, then $\sigma_1 \cup \sigma_2$ is a unifier for $T_L \equiv_q^? T_R$, where σ_1 and σ_2 are unifiers for $T_1 \equiv_q^? T'_1$ and $T_2 \equiv_q^? T'_2$, respectively.*

PROOF. Observe that qualified type variables with the same variable name but different qualifiers are treated as different type variables. Therefore, given a choice type $D\langle T_l, T_r\rangle$, it is always the case that $vars(T_l) \cap vars(T_r) = \varnothing$, by the definition of qualification. Specifically, the qualifier of every type variable in T_l will contain the selector D, while the qualifier of every type variable in T_r will contain the selector $\tilde{D}$. Since this property holds for both choice types in the lemma, the unification subproblems do not share any common type variables and are therefore independent. □

Next we show that splitting a type variable is variable independent. This means that when more than one type variable is splittable, we will achieve an equivalent unifier no matter which type variable we choose to split.

LEMMA 7.3 (VARIABLE INDEPENDENCE). *Let $T_L = D_1\langle T_1, T_2\rangle$ and $T_R = D_2\langle T'_1, T'_2\rangle$. Assume $a_q, b_r \in splittable(T_L)$, where qualifiers q and r do not contain selectors in dimension D_2. Let $\theta_a = \{a_q \mapsto D_2\langle a_{D_2q}, a_{\tilde{D}_2q}\rangle\}$ and $\theta_b = \{b_r \mapsto D_2\langle b_{D_2r}, b_{\tilde{D}_2r}\rangle\}$. Then $unify(T_L\theta_a, T_R) = unify(T_L\theta_b, T_R)$.*

$$
\begin{array}{r}
D_1\langle \underset{split\downarrow}{a_q}, T_1\rangle \equiv^?_q D_2\langle T_2, D_3\langle T_3, T_4\rangle\rangle \\
D_1\langle \underset{hoist\downarrow}{D_2}\langle a_{D_2q}, a_{\tilde{D}_2q}\rangle, T_1\rangle \equiv^?_q D_2\langle T_2, D_3\langle T_3, T_4\rangle\rangle \\
D_2\langle D_1\langle a_{D_2q}, T_1\rangle, D_1\langle a_{\tilde{D}_2q}, T_1\rangle\rangle \equiv^?_q D_2\langle T_2, D_3\langle T_3, T_4\rangle\rangle \\
\swarrow \qquad\qquad \searrow \\
D_1\langle a_{D_2q}, T_1\rangle \equiv^?_q T_2 \qquad D_1\langle a_{\tilde{D}_2q}, T_1\rangle \equiv^?_q D_3\langle T_3, T_4\rangle \\
\vdots \\
D_1\langle a_{\tilde{D}_2D_3q}, T_1\rangle \equiv^?_q T_3 \\
D_1\langle a_{\tilde{D}_2\tilde{D}_3q}, T_1\rangle \equiv^?_q T_4
\end{array}
$$

$$
\begin{array}{r}
D_1\langle \underset{split\downarrow}{a_q}, T_1\rangle \equiv^?_q D_2\langle T_2, D_3\langle T_3, T_4\rangle\rangle \\
D_1\langle \underset{hoist\downarrow}{D_3}\langle a_{D_3q}, a_{\tilde{D}_3q}\rangle, T_1\rangle \equiv^?_q D_2\langle T_2, D_3\langle T_3, T_4\rangle\rangle \\
D_3\langle D_1\langle a_{D_3q}, T_1\rangle, D_1\langle a_{\tilde{D}_3q}, T_1\rangle\rangle \equiv^?_q D_2\langle T_2, \underset{\downarrow hoist}{D_3}\langle T_3, T_4\rangle\rangle \\
D_3\langle D_1\langle a_{D_3q}, T_1\rangle, D_1\langle a_{\tilde{D}_3q}, T_1\rangle\rangle \equiv^?_q D_3\langle D_2\langle T_2, T_3\rangle, D_2\langle T_2, T_4\rangle\rangle \\
\swarrow \qquad\qquad \searrow \\
D_1\langle a_{D_3q}, T_1\rangle \equiv^?_q D_2\langle T_2, T_3\rangle \qquad D_1\langle a_{\tilde{D}_3q}, T_1\rangle \equiv^?_q D_2\langle T_2, T_4\rangle \\
\vdots \qquad\qquad\qquad\qquad \vdots \\
D_1\langle a_{D_2D_3q}, T_1\rangle \equiv^?_q T_2 \qquad D_1\langle a_{D_2\tilde{D}_3q}, T_1\rangle \equiv^?_q T_2 \\
D_1\langle a_{\tilde{D}_2D_3q}, T_1\rangle \equiv^?_q T_3 \qquad D_1\langle a_{\tilde{D}_2\tilde{D}_3q}, T_1\rangle \equiv^?_q T_4
\end{array}
$$

Fig. 12. Demonstration of choice independence.

PROOF. After splitting a type variable a_q (or b_r) in T_L, the newly formed choice type must be hoisted to the top so that the unification problem can be decomposed by alternatives. After applying this series of hoists, we obtain a new type T'_L with a choice type in dimension D_2 at the top level. The lemma depends crucially on the fact that no matter which type variable we split, T'_L will be the same, and therefore the result of the unfication will be the same. This is true because the process of hoisting the new D_2 choice type will essentially cause every other type variable in T_L to be split in dimension D_2. This process is described below.

By the definition of *splittable*, the path from the top of T_L to a_q (or b_r) consists of only choice types. For each choice type $D_i\langle\ldots\rangle$ along this path, we must apply *hoist* once in order to lift the D_2 choice type outward one level. Without loss of generality, assume that the D_2 choice type is in the left alternative, so $D_i\langle D_2\langle T_1, T_2\rangle, T_3\rangle$. After applying *hoist*, we have $D_2\langle D_i\langle T_1, T_3\rangle, D_i\langle T_2, T_3\rangle\rangle$. Since T_3 was copied into both alternatives of the D_2 choice type, every type variable in the first T_3 will be qualified by D_2 while every type variable in the second T_3 will be qualified by $\tilde{D}_2$. In this way, the process of hoisting the D_2 choice type to the top level will cause every other type variable to be split into two type variables qualified by selectors for dimension D_2. □

The process described in the proof of Lemma 7.3 is illustrated in Figure 11 with a small example. In this example, $T_L = D_1\langle a_q, D_3\langle T_1, b_r\rangle\rangle$ and $T_R = D_2\langle T_2, T_3\rangle$, where q does not contain qualifiers in D_2 or D_3 and r does not contain a qualifier in D_2. In the top case, we split a_q in dimension D_2, while in the bottom, we split b_r. Observe how type variables are copied and qualified when a choice type is hoisted over them.

Just as it does not matter which splittable type variable we choose, it does not matter which dimension we choose to split it in, as long as the type variable is not already qualified by that dimension.

LEMMA 7.4 (CHOICE INDEPENDENCE). *Let* $T_L = D_1\langle T_1, T_2\rangle$ *and* $T_R = D_2\langle T_1', T_2'\rangle$. *Assume* $D_m, D_n \in$ *choices*(T_R) *and* $a_q \in$ *splittable*(T_L), *where qualifier* q *does not contain selectors in* D_m *or* D_n. *Let* $\theta_1 = \{a_q \mapsto D_m\langle a_{D_m q}, a_{\tilde{D_m} q}\rangle\}$ *and* $\theta_2 = \{a_q \mapsto D_n\langle a_{D_n q}, a_{\tilde{D_n} q}\rangle\}$. *Then* $\mathit{unify}(T_L\theta_1, T_R) = \mathit{unify}(T_L\theta_2, T_R)$.

PROOF. Assume without loss of generality that we split a_q in dimension D_m. If $D_m = D_2$, we can make progress by hoisting the newly created choice type in D_m to the top of T_L and decomposing the resulting unification problem by alternatives. Otherwise, we will have to also hoist the choice in D_m to the top of T_R, then decompose. Either way, this will result in at least two new type variables, $a_{D_m q'}$ (in the left decomposition of T_L) and $a_{\tilde{D_m} q''}$ (in the right decomposition of T_L). Since q did not contain a selector in D_n, there is no choice type on the path from a_q to the root of T_L, so neither q' nor q'' will contain a selector in D_n, which might otherwise have been introduced during the hoisting process. However, since a choice type in D_n still exists in at least one of the two subproblems, we will have to split one or both of $a_{D_m q'}$ and $a_{\tilde{D_m} q''}$ in dimension D_n in order to complete the unification. Therefore, since we must eventually split the original a_q in both D_m and D_n, proving choice independence is equivalent to proving that the order in which we perform these splits does not affect the unification result.

Suppose we perform both splits before doing any decompositions. If we split a_q in D_m first and then D_n, a_q will be replaced by the type T_a below. If we split a_q in D_n and then D_m, it will be replaced by the type T_a'.

$$T_a = D_m\langle D_n\langle a_{D_m D_n q}, a_{D_m \tilde{D_n} q}\rangle, D_n\langle a_{\tilde{D_m} D_n q}, a_{\tilde{D_m}\tilde{D_n} q}\rangle\rangle$$
$$T_a' = D_n\langle D_m\langle a_{D_m D_n q}, a_{\tilde{D_m} D_n q}\rangle, D_m\langle a_{D_m \tilde{D_n} q}, a_{\tilde{D_m}\tilde{D_n} q}\rangle\rangle$$

It is easy to see that $T_a \equiv T_a'$ by the equivalence rules in Figure 5. We can transform T_a into T_a' by applying the C-C-SWAP rules to each alternative, then applying the C-C-MERGE rules to eliminate the dominated choice types. Since T_a and T_a' are equivalent, then $T_L' = T_L\{a_q \mapsto T_a\}$ and $T_L'' = T_L\{a_q \mapsto T_a'\}$ are also equivalent, so the results of unifying $T_L' \equiv_q^? T_R$ and $T_L'' \equiv_q^? T_R$ will be the same. □

The process described in the proof of Lemma 7.4 is illustrated in Figure 12. In this example, $T_L = D_1\langle a_q, T_1\rangle$ and $T_R = D_2\langle T_2, D_3\langle T_3, T_4\rangle\rangle$, where the qualifier q does not contain qualifiers in dimensions D_2 or D_3. In the top case we split a_q into a choice type in dimension D_2, eventually yielding three unification subproblems. In the bottom case we split a_q in dimension D_3, eventually yielding four subproblems. (The vertical ellipses in each of these derivations represent further splitting the type variable in dimension D_3, hoisting this choice to the top level, and decomposing by alternatives.) However, observe that the subproblem on the left branch of the top case is equivalent to the two subproblems in the bottom case that have T_2 on their RHS. In order to obtain the subproblems in the bottom case, we can use choice idempotency to rewrite T_2 to $D_3\langle T_2, T_2\rangle$, then decompose by alternatives.

Since the hoisting operation only restructures a type in a semantics-preserving way, its correctness is obvious.

Our unification algorithm is terminating through decomposition that eventually produces calls to Robinson's unification algorithm (which is terminating). The only challenge to termination is that the splitting of type variables introduces new choice types to the types that are being unified. However, two facts ensure that this does not prevent termination: (1) a variable can only be split into a choice type whose dimension occurs in the type being unified

against and (2) immediately after a split is performed the new choice type is hoisted and decomposed, producing two subtasks that are each smaller than the original task.

The qualified unification algorithm is sound, meaning the mappings it produces always unify its arguments. It is also complete and most general, which means that if the two types are unifiable, then the algorithm will return the most general mapping that unifies them. We express each of these results in the following theorems.

THEOREM 7.5 (SOUNDNESS). *If* $unify(T_1, T_2) = \sigma$, *then* $T_1\sigma \equiv T_2\sigma$.

THEOREM 7.6 (COMPLETE AND MOST GENERAL). *If* $T_1\sigma \equiv T_2\sigma$, *then* $unify(T_1, T_2) = \sigma'$ *where* $\sigma = \sigma'' \circ \sigma'$ *for some* σ''.

A proof of Theorem 7.5 is given in the appendix in Section A.2.1.

In order to prove the correctness of CT-unification, we must relate the above theorems on qualified unification to the problem of variational unification. To do this, we must first establish the relationships between the *comp* function, the qualifiers, the unification problem, and the qualified unification problem.

The following lemma states the expectations for the *comp* function, which transforms a mapping from a single qualified type variable into a mapping from an unqualified type variable to a partially completed type. The lemma is proved in the appendix in Section A.2.2, demonstrating that *comp* is correct.

LEMMA 7.7. *Given a mapping* $\{a_q \mapsto T'\}$, *if* $T = comp(q, T', b)$ *is the completed type (where* b *is fresh), then* $\lfloor T \rfloor_q = \lfloor T' \rfloor_q$. *More generally, given* $\{a_{q_1} \mapsto T_1, \ldots, a_{q_n} \mapsto T_n\}$, *if* $T = comp(q_1, T_1, comp(q_2, T_2, \ldots comp(q_n, T_n, b)\ldots))$, *then for every* $q_i \in \{q_1 \ldots q_n\}$, *we have* $\lfloor T \rfloor_{q_i} = \lfloor T_i \rfloor_{q_i}$.

Using this result, we can prove the correctness of the completion process. Given a solution to a qualified unification problem, completion produces a solution to the original unqualified version. The lemma below states the expectation of completion with respect to the selection semantics. It is also proved in the appendix, in Section A.2.3.

LEMMA 7.8 (COMPLETION). *Given a CT-unification problem* $T_L \equiv^? T_R$ *and the corresponding qualified unification problem* $T'_L \equiv^?_q T'_R$, *if* σ_Q *is a unifier for* $T'_L \equiv^?_q T'_R$ *and* σ_U *is the unifier attained by completing* σ_Q, *then for any super-complete decision* δ, $\lfloor T_L\sigma_U \rfloor_\delta \equiv \lfloor T'_L\sigma_Q \rfloor_\delta$ *and* $\lfloor T_R\sigma_U \rfloor_\delta \equiv \lfloor T'_R\sigma_Q \rfloor_\delta$.

Completion also preserves principality since the *comp* function adds fresh type variables everywhere except at the leaf addressed by the path q (maximizing generality), and the principal type inferred during qualified unification is inserted directly at q.

The following theorem generalizes Lemma 7.8, stating that through qualification and completion, we can solve CT-unification problems. We call this process *variational unification*.

THEOREM 7.9. *Given a CT-unification problem* U *and the corresponding qualified unification problem* Q, *if* σ_Q *is a unifier for* Q, *then we can attain a unifier* σ_U *for* U *through the process of completion.*

Variational unification is sound, complete, and most general since the underlying qualified unification algorithm has these properties and since completion preserves principality, a fact that follows directly from Lemma 7.7.

7.4. Time Complexity of the Unification Algorithm

Solving the unification problem U consists of three steps: (1) transforming U into the corresponding qualified unification problem Q, (2) solving Q with the qualified unification algorithm *unify*, and (3) transforming the qualified unifier into the variational unifier through

completion. To determine the time needed to solve U, we will consider the time complexity of each step in turn. As before, we use T_L and T_R to denote the LHS and RHS of U. We use σ_U and σ_Q to denote the unifier for U and Q, respectively. The size of a type T, denoted by $|T|$, is the number of nodes in its AST (as defined by the grammar in Section 3.1). We assume that $|T_L| = l$ and $|T_R| = r$. The size of a unification problem is the sum of the sizes of the types being unified.

The process of transforming U to Q qualifies each type variable in U. This can be achieved by a top-down traversal of the ASTs of T_L and T_R. Thus, the complexity of this process is $O(l + r)$. Note that the resulting qualified problem Q is the same size as the original U.

For the second step of solving Q, we do a worst-case complexity analysis. For simplicity, assume that the internal nodes of T_L and T_R are all choice types. Then the worst case for unification is when $\mathit{choices}(T_L) \cap \mathit{choices}(T_R) = \varnothing$. When T_L and T_R have no choices in common, we proceed by (1) splitting a type variable in one of the types, say T_L, into a choice type in the dimension of the root choice of the other type, T_R; (2) hoisting the new choice type to the root of T_L; and (3) decomposing the problem by alternatives. Splitting and hoisting the new choice type increases the size of the LHS to $1 + 2l$: 1 for the new choice type plus $2l$ for the copy of T_L in each alternative with extended qualifications on its type variables. The splitting and hoisting process can be performed in $O(l)$ time by introducing the new choice type, copying T_L into each alternative, and then traversing each alternative, qualifying the type variables accordingly.

After decomposing the problem by alternatives, we are left with two smaller subproblems $T_{L_1} \equiv_q^? T_{R_1}$ and $T_{L_2} \equiv_q^? T_{R_2}$. We know that $|T_{L_1}| = |T_{L_2}| = |T_L|$ since T_{L_1} and T_{L_2} are just copies of T_L with different type variable qualifiers. Moreover, $|T_{R_1}| + |T_{R_2}| = |T_R| - 1$ since T_{R_1} and T_{R_2} are the left and right branch of the root node of T_R, respectively. The split-hoist-decompose process will be recursively applied to the subproblems $T_{L_1} \equiv_q^? T_{R_1}$ and $T_{L_2} \equiv_q^? T_{R_2}$. After two more decompositions, there will be four unification subproblems. Since there are $(r-1)/2$ internal nodes, there will be $(r-1)/2$ decompositions, and since each decomposition takes $O(l)$ time, the whole decomposition takes $O(l \cdot (r-1)/2)$ time.

We can also observe that each decomposition by alternatives creates two subproblems from one. This will result in $(r+1)/2$ subproblems, one from the decomposition corresponding to each choice node in the tree. Based on the decomposition process, each resulting subproblem is either of the form $T'_L \equiv_q^? g$ or $T'_L \equiv_q^? a_q$, where T'_L differs from T_L only in type variable qualifications, g is a ground type, and a_q is a qualified type variable. Based on cases (2) and (7) of the unification algorithm, each final subproblem therefore takes $O(|T'_L|) = O(|T_L|) = O(l)$ to solve. Thus the time needed to solve all subproblems is $O(l \cdot (r+1)/2)$.

Summing the time needed for decomposition, $O(l \cdot (r-1)/2)$, and the time needed for solving the resulting unification problems, $O(l \cdot (r+1)/2)$, the time complexity of solving Q is $O(lr)$.

Finally, we consider the complexity of the third step of the unification process, transforming the solution σ_Q for Q into a solution σ_U for U through the process of completion. Again, we perform a worst-case analysis. Completion is performed by folding the mappings in σ_Q with the function *comp*. We can establish an upper bound on the number of mappings in σ_Q by following the decomposition process in the previous step and counting the number of potential type variables. At the end of this process we have at most $(r-1)/2$ subproblems of the form $T'_L \equiv_q^? T$. Each T'_L is of size l and contains at most $(l+1)/2$ type variables at the leaves; each T is either a type variable or a ground plain type. Therefore, after some simplification, σ_Q contains at most $(1/2)(r-1)(l+1)$ mappings.

If we think of the completion process as incrementally building up a tree of nested choices that describe the result type T, then each mapping $a_{q_i} \mapsto T_i \in \sigma_Q$ essentially describes a leaf in that tree. Applying *comp* to such a mapping constitutes traversing T according to the path described by q_i, possibly generating at most one new choice type and one new type variable (if this is the first traversal along this path) at each step of the way; this takes $O(|q_i|)$ time,

where $|q_i|$ is the length of the qualifier. The length of the qualifier is in turn bounded by the total number n of dimensions present in the unification problem. An upper bound on n can be expressed in terms of l and r as $(l-1)/2+(r-1)/2$ since there is at most one dimension name for each internal node in the original types T_L and T_R. Finally, since a single completion step takes time $O(n) = O(l+r)$ and we will perform $O(lr)$ completion steps (one for each mapping), the total time for completion is $O(l^2r + lr^2)$.

Summing these three steps, we see that the completion step dominates the others, so the unification of variational types takes cubic time, in the worst case, with respect to the size of the types. When unifying types that contain choice types in the same dimension, we can expect the complexity of unification to be much lower.

8. TYPE INFERENCE ALGORITHM

Although the unification algorithm for VLC differs significantly from the Robinson unification algorithm, the type inference algorithm is only a simple extension of algorithm $\mathcal{W}$ for lambda calculus [Damas and Milner 1982]. We call this algorithm *infer* and its type is given below.

$$infer : \Lambda \times \Gamma \times e \to \sigma \times T$$

The function takes three arguments: the two environments maintained in the typing rules (the **share**-bound variable environment Λ and the typing environment Γ) and the expression to type. It returns a type substitution and the inferred type.

The cases of the *infer* algorithm can be derived from the typing rules in Section 3.2. The cases for choices and applications are given below.

$infer(\Lambda, \Gamma, D\langle e_1, e_2\rangle) =$
  $(\sigma_1, T_1) \leftarrow infer(\Lambda, \Gamma, e_1)$
  $(\sigma_2, T_2) \leftarrow infer(\Lambda, \Gamma\sigma_1, e_2)$
  return $(\sigma_2 \circ \sigma_1, D\langle T_1, T_2\rangle)$

$infer(\Lambda, \Gamma, e_1\ e_2) =$
  $(\sigma_1, T_1) \leftarrow infer(\Lambda, \Gamma, e_1)$
  $(\sigma_2, T_2) \leftarrow infer(\Lambda, \Gamma\sigma_1, e_2)$
  $\sigma \leftarrow unify'(T_1\sigma_2, T_2 \to a)$ *{- a is a fresh type variable -}*
  return $(\sigma \circ \sigma_2 \circ \sigma_1, a\sigma)$

On a choice, we determine the alternative types in the result by inferring the type of each alternative expression. Note that we apply the mapping produced by inferring the type of e_1 to the typing environment used to infer the type of e_2. This ensures that the result types and mappings will be consistent. Finally, the result mapping is just a composition of the mappings produced during type inference of the two alternatives.

The *infer* algorithm types applications as in $\mathcal{W}$, except replacing the unification algorithm with our own variational unification algorithm (and propagating the additional environments). We use $unify'$ to represent the combined qualification, unification, and completion process. That is, first the type variables in T_1 and T_2 are qualified, then *unify* is invoked on the transformed types, and finally the resulting mapping is completed to produce σ, the solution to the original unqualified unification problem.

The remaining cases are similarly straightforward. Abstractions and λ-bound variables are exactly as in $\mathcal{W}$, while **share** expressions and **share**-bound variables can be derived from the typing rules in the same way as the choice case above.

The following theorem expresses the standard property of soundness for the variational type inference algorithm.

THEOREM 8.1 (TYPE INFERENCE IS SOUND). $infer(\Lambda, \Gamma, e) = (\sigma, T) \implies \Lambda, \Gamma\sigma \vdash e : T$.

PROOF. The *infer* algorithm is directly derived from the typing rules and based on algorithm $\mathcal{W}$, which is sound. The only challenge to soundness comes from the divergence from $\mathcal{W}$ on applications, where we replace the standard unification algorithm with $unify'$. However, since variational unification is also sound per Theorem 7.9, this property is preserved. □

The type inference algorithm also has the principal type property, which follows from Theorems 7.6 and 7.9.

THEOREM 8.2 (TYPE INFERENCE IS COMPLETE AND PRINCIPAL). *For every mapping* σ *and type* T *such that* $\Lambda, \Gamma\sigma \vdash e : T$*, there exists a* σ' *and* T' *such that* $infer(\Gamma, e) = (\sigma', T')$ *where* $\sigma = \sigma'' \circ \sigma'$ *for some* σ'' *and* $T = T'\sigma'''$ *for some* σ'''.

These results are important because they demonstrate that properties from other type systems can be preserved in the context of variational typing.

9. EFFICIENCY

For any static variation representation (such as the choice calculus) applied to a statically-typed object language, there exists a trivial typing algorithm: generate every program variant, then type each one individually using the non-variational type system of the object language. We call this the "brute-force" strategy. There are two significant advantages of a more integrated approach using variational types. The first is that we can characterize the variational structure of types present in variational software—this is useful for aiding understanding of variational software and informing decisions about which program variant to select. The second is that we can gain significant efficiency improvements over the brute-force strategy. Due to the combinatorial explosion of program variants as we add new dimensions of variation, separately inferring or checking the types of all program variants quickly becomes infeasible. In this section we describe how variational type systems, and our type system for VLC in particular, can increase the efficiency of type inference for variational programs, making typing possible for massively variable systems. We do this in two ways: by analytically characterizing the opportunities for efficiency gains, and by demonstrating these gains experimentally.

9.1. Analytical Characterization of Efficiency Gains

Although we have considered only binary dimensions so far, we assume in this discussion that the variational type system has been extended to support arbitrary n-ary dimensions. While this extension is not interesting from a technical perspective, it is important for practical use and accentuates the potential for efficiency gains.

An important observation is that the *worst-case* performance of any variational type system is guaranteed to be no better than the brute-force strategy, assuming the variation representation is sufficiently general. Consider the following VLC expression.

$$A\langle B\langle e_1, e_2\rangle, B\langle e_3, e_4\rangle\rangle$$

If e_1, e_2, e_3, and e_4 contain no common parts that can be factored out, there is simply no improvement to be made over the brute-force strategy. We must type each of the four expressions separately, and the type of each one provides no insight into the types of others. Fortunately, we expect there to be many more opportunities for improvement in actual software. In this section, we describe the two basic ways that variational typing can save over the brute-force strategy, characterizing the efficiency gains by each. Since these patterns are expected to be ubiquitous in practice, variational typing can likewise be expected to be much more efficient.

The first opportunity for efficiency gains arises because choices capture variation *locally*. This allows the type system to reuse the types inferred for the common context of the alternatives in a choice. Suppose we have a choice $D\langle e_1, \ldots, e_n\rangle$ in a non-variational context C.

Conceptually, a context is an expression with a hole; we can fill that hole with the choice above to produce the overall expression, which we write as $C[D\langle e_1,\ldots,e_n\rangle]$. Our algorithm types the contents of C only once, whereas the brute-force strategy would type each $C[e_i]$ separately, typing C a total of n times. While the work performed on C by our algorithm is constant, the extra work performed by the brute-force strategy obviously grows multiplicatively with the size of each new dimension of variation. We can maximize the benefits of choice locality gains by ensuring that choices are *maximally factored*. In our previous work we say that such expressions are in *choice normal form*, and we provide a semantics-preserving transformation to achieve this desirable state [Erwig and Walkingshaw 2011].

The second, more subtle opportunity involves the typing of applications between two choices, for example, $A\langle e_1,\ldots,e_n\rangle\ B\langle e'_1,\ldots,e'_m\rangle$. Since the brute-force strategy considers every variant individually, it must unify the type of every alternative in the first choice with the type of every alternative in the second choice, for a total of $n\cdot m$ unifications. The ability to see all variants together provides substantial opportunity for speed-up if several alternatives in either choice have the same type. For example, if the alternatives of the choice in dimension A have $k < n$ unique types and the alternatives of the choice in B have $l < m$ unique types, then the type inference algorithm must invoke unification at most $k \cdot l$ times (and often less, depending on the structure of the types). Since we expect it to often be the case that all alternatives of a choice have the *same* type (consider varying the values of constants, the names of variables, or only the implementation of a function), this offers a dramatic opportunity for efficiency gains.

9.2. Experimental Demonstration

In this section we continue the efficiency discussion by demonstrating the performance of our variational type inference algorithm, *infer*, on several example VLC expressions. For reference, we compare these times to the time needed to type a single variant from the expression, and (where feasible) the time needed to type all variants using the brute-force strategy. These results are not intended as a rigorous or real-world experimental evaluation of the variational type inference algorithm. Rather, they are intended as a simple demonstration of the efficiency gains described in the previous subsection, and as a vehicle to further this discussion.

We have developed a prototype in Haskell that implements the ideas and algorithms presented in this paper. The prototype consists of three parts: the variational type normalizer, the equational unification algorithm, and the type inference algorithm. The prototype implements most of the features described in this paper. One exception is that the second opportunity for efficiency gains, described in the previous subsection, is exploited only in the special case when all variants of a choice have the same type.

Throughout the first part of this discussion, we will be referring to the expressions and results in Figure 13. The leftmost column names each expression and the next column defines it. The *dims* column indicates the number of dimensions in the expression. The *variants* column indicates the total number of variants, which can be calculated by multiplying the arity of each of the dimensions. The timing results are given in the final three columns. The *one* column indicates the time needed to infer the type of a *single* program variant. This is intended as a reference point for comparison with the other two timing results. The *brute* column gives the time to infer the type of each variant separately using the brute-force strategy, and *vlc* gives the time taken by *infer* to infer a variational type for the expression.

All times are calculated within our prototype. In the absence of variation (when inferring types for *one* and *brute*), the prototype reduces to a standard implementation of algorithm $\mathcal{W}$. The typing environment is seeded with boolean and integer values that map to constant types `Bool` and `Int`, and several simple functions like `id`, `not`, `succ`, and `even` that map to the expected types. The function `if` has type $\texttt{Bool} \to a \to a \to a$.

	expression	dims	variants	one	brute	vlc
e_1	$A\langle\lambda$x.3, λx.true$\rangle$ $B\langle$3,5$\rangle$	2	4	2.6	10.2	10.1
e_2	$A\langle\lambda$x.3, λx.true$\rangle$ $B\langle$3,5,7,9$\rangle$	2	8	2.5	20.4	10.2
e_3	if true e_1 e_1'	4	16	21.2	334.7	34.3
e_4	$A\langle B\langle$succ, λx.true$\rangle$, $B\langle\lambda$x.3, not$\rangle\rangle$ $B\langle$3, true$\rangle$	2	4	2.6	10.1	15.7
e_5	id id id e_4	2	4	31.9	125.1	63.0
e_6	if true e_4 e_4'	4	16	24.0	380.5	55.0

Fig. 13. Running times of type inference strategies on several examples. Each test was run 200,000 times on a 2.8GHz dual core processor with 3GB of RAM. All times are in seconds.

The first group of expressions demonstrates some basic relationships between the number and arity of dimensions and the potential efficiency gains of variational type inference. In e_1 we present a simple unification problem with an opportunity for sharing (both alternatives of the B choice have type `Int`). Since the number of variants is so small, the overhead of *infer* negates the gains made by sharing, and the algorithm performs equivalently to the brute-force strategy. However, this quickly changes as we add variants and additional context. In e_2 we have doubled the number of variants by increasing the number of alternatives in the B dimension from 2 to 4. While the running time for the brute-force strategy correspondingly doubles, variational type inference does not since the new alternatives can also be shared. Finally, in e_3 we double the number of dimensions to increase the number of variants by a factor of four. The expression e_1' is identical to e_1, except with unique dimension names. This example also adds some additional, unvaried context (`if True`). Now we can see the exponential explosion of the brute-force strategy (which must also type the common context 16 times), while *infer* scales essentially linearly with respect to the size of the expression. We can also observe that the ratio of overhead for *infer*, relative to the reference single-variant inference time, decreases as we increase the size of the expression.

In the second group we begin with a more complex variational structure with no opportunities for sharing, e_4. As expected, *infer* performs worse than brute-force due to the overhead. With e_5, however, we demonstrate how even a very small amount of common context can tip the scale back in *infer*'s favor. If we again duplicate the initial expression and rename the dimension names, as in e_6, we introduce an opportunity for sharing, allowing *infer* to scale nicely while brute-force does not.

Next we analyze the impact of a difficult case for our algorithm that we call *cascading choices*. This can occur when we have a long sequence of choices, each in a different dimension, connected by applications. If there are few opportunities for sharing, the result type produced from the first unification can be expanded by the second, third, and so on, potentially building up a result type exponentially and making each successive unification more expensive than the last.

Figure 14 demonstrates the performance of *infer* and the brute-force algorithm on expressions designed to induce the cascading choice problem. In the left graph, the x-axis indicates the number of dimensions in the expression, and the y-axis gives the running time on a logarithmic scale. The leftmost expression with 14 dimensions has 16384 variants and produces a result type with 14335 different variants. We can observe that the running time of *infer* is exponential with regard to the number of dimensions. However, it still performs slightly better than the brute-force strategy because it takes advantage of the few opportunities for sharing available.

While *infer* is sensitive to the number of dimensions in expressions inducing the cascading choice problem, it is less sensitive to the overall size of the expression. This is in stark contrast to the brute-force strategy, as illustrated in the right graph in Figure 14. Here, we fix the length of cascading choices at 21 but increase the size of the expression by making the alternatives in each choice more complex. The x-axis shows this size (in number of AST nodes)

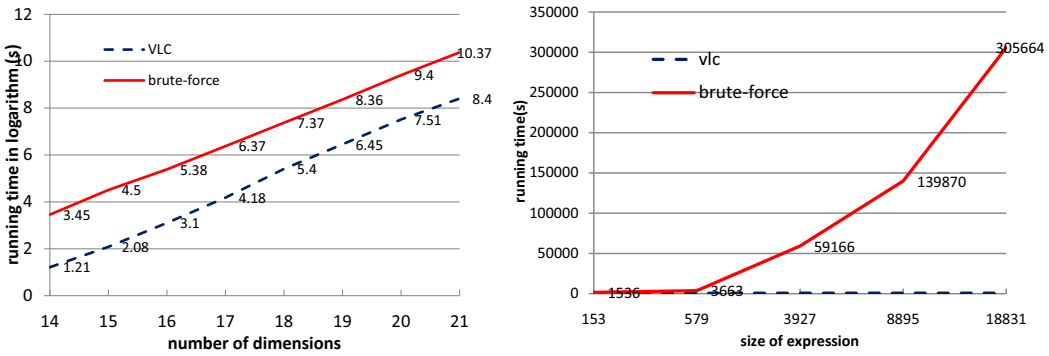

Fig. 14. Performance comparison for the cascading choice problem.

size	dims	c/d	a/s	nesting	cascading	one	brute	vlc
569	27	5.38	0.070	11(1)	11(1)	0.0011	148504	0.52
3505	57	6.61	0.279	12(1)	11(2)	0.0236	-	0.57
8153	168	6.21	0.250	12(4)	12(4)	0.0583	-	2.14
9429	215	6.67	0.218	12(7)	12(5)	0.0510	-	3.16
29481	681	6.87	0.210	12(25)	12(17)	0.154	-	10.16
61345	1434	7.05	0.203	12(56)	12(37)	0.321	-	21.67
213521	4983	7.03	0.203	12(183)	12(119),13(3)	1.10	-	76.98
429586	10002	7.08	0.202	17(2),12(287)	19(1),12(229)	2.17	-	142.33

Fig. 15. Running times of type inference for large expressions (in seconds).

and the y-axis shows the running time in seconds. We observe that the brute-force strategy grows sharply as the size of the alternatives increases since each will be typed several times. This additional work will be shared in *infer*, however, and so the running time grows much less (increasing from 338 seconds to 461).

Finally, in Figure 15, we demonstrate the efficiency of type inference on several large, randomly generated expressions. These expressions are generated in several steps. First, we add several functions and their corresponding types to an initial environment. Then we manually build up a library of small (potentially variational) expressions and add these to the environment. We use these seeded expressions as building blocks for randomly constructing larger expressions by picking a random function from the environment, picking random arguments that will satisfy its type, possibly changing the dimension names, then joining these expressions with applications. Finally, we add this new expression back into the environment and repeat until the environment contains expressions of the desired size and complexity.

The table reports the running time for many large expressions. It gives the size of each expression as the number of AST nodes and the number of contained dimensions. Each dimension is binary, so an expression with d dimensions will describe 2^d total variants. The next four columns characterize the composition and structure of the expression. We indicate the ratio c/d of choices to dimensions, and the ratio a/s of application nodes to the size of the expression. In general, we would expect a higher ratio of application nodes to present a greater challenge for the inference algorithm (since unification must be invoked more often). The column *nesting* indicates the deepest choice-nestings in the expression, where $d(n)$ indicates that the nesting depth d occurs n times. Similarly, the column *cascade* indicates the longest occurrences of cascading choices, as described above. In the last three columns we give the time required to infer the type of a single variant, to infer the types of all vari-

ants using the brute-force strategy, and to infer a variational type using *infer*. Note that it is impossible to apply the brute-force approach to all but the first of these expressions.

These results demonstrate the feasibility of variational type inference on very large expressions. Our results for type inference are consistent with those for type checking demonstrated by Thaker et al. [2007]. While usually much larger in size, we would expect real-world software to be considerably less complex. For example, in an analysis of real variational software implemented with the AHEAD framework [Batory et al. 2004], Kim et al. [2008] found a maximum nesting depth of just 3 and an average depth of 1.5.

10. RELATED WORK

There are many different ways of encoding variability in software. In VLC, variation points are embedded directly into lambda calculus terms with choices. In Section 10.1 we compare this annotative approach to other ways of encoding variation and motivate our decision to base our type system on annotative variation.

The rest of this section addresses work related to variational types, VLC's type system, typing variational programs, and the variational type inference algorithm. This work falls roughly into two categories. In Section 10.2 we discuss related theoretical work in the area of programming languages and type systems. In Section 10.3 we relate our approach to other work done in the area of software product lines.

10.1. Approaches to Variation Management

In general, there are three main ways to manage software variation, which we will refer to in the following as (1) *annotative*, (2) *compositional*, and (3) *metaprogramming*.

(1) In the annotative approach, object language code is varied in-place through the use of a separate annotation metalanguage. Annotations delimit code that will be included or not in each program variant. When selecting a particular variant from an annotated program, the annotations and any code not associated with that variant are removed, producing a plain program in the object language. The most widely used annotative variation tool is the C Preprocessor (CPP) [GNU Project 2009], which supports static variation through its conditional compilation constructs (`#ifdef` and its relatives).

(2) The compositional approach emphasizes the separation of variational software into its component *features* and a shared *base program*, where a feature represents some functionality that may be included or not in a particular variant. Variants are generated by selectively applying a set of features to the base program. This strategy is usually applied in the context of object-oriented languages and relies on language extensions to separate features that cannot be captured in traditional classes and subclasses. For example, inheritance might be supplemented by mixins [Bracha and Cook 1990; Batory et al. 2004], aspects [Elrad et al. 2001; Mezini and Ostermann 2003], or both [Mezini and Ostermann 2004]. Relationships between the features are described in separate configuration files [Batory et al. 2004], or in external documentation, such as a feature diagram [Kang et al. 1990]. These determine the set of variants that can be produced.

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

Annotative approaches provide a desirable foundation for developing general strategies for extending existing analyses to variational programs. By providing a language independent, highly structured, and visible model of variation, annotative approaches separate variational concerns from the object language. They can be applied to multiple different object languages,

and they enable the direct manipulation and analysis of the variation within the software. These qualities feature prominently in the type system for VLC presented in this paper, allowing us to represent variation explicitly not only in the language of expressions, but also in the language of types, and to correlate the two.

10.2. Programming Languages and Type Systems

Choice types are in some ways similar to variant types [Kagawa 2006]. Variant types support the uniform manipulation of a heterogeneous collection of types. A significant difference between the two is that choices (at the expression level) contain all of the information needed for inferring their corresponding choice type. Values of variant types, on the other hand, are associated with just one label, representing one branch of the larger variant type. This makes type inference very difficult. A common solution is to use explicit type annotations; whenever a variant value is used, it must be annotated with a corresponding variant type. Typing VLC expressions does not require such annotations.

Choice types are also somewhat similar to union types [Dezani-Ciancaglini et al. 1997]. A union type, as its name suggests, is a union of simpler types. For example, a function `f` might accept as arguments the union of types `Int` and `Bool`. Function application is then well typed if the argument's type is an element of the union type; so, `f` could accept arguments of type `Int` or type `Bool`. The biggest difference between union types and choice types is that union types are comparatively unstructured. In VLC, choices can be synchronized, allowing functions to provide different implementations for different argument types, or for different sets of functions to be defined in the context of different argument types. With union types, an applied function must be able to operate on all possible values of an argument with a union type. A major challenge in type inference with union types is union elimination, which is not syntax directed and makes type inference intractable. Therefore, as with variant types, syntactic markers are needed to support type inference.

Type conditions are an extension to parametric polymorphism in the presence of subtyping that have been studied in the contexts of both the Java generics system [Huang et al. 2007] and C++ templates [Dos Reis and Stroustrup 2006]. They can be used to conditionally include data members and methods into a class only when the type parameters are instantiated with types that satisfy the given conditions (for example, that the type is a subtype of a certain class). Often this can be used to produce similar effects to the C Preprocessor, but in a way that can be statically typed. Type conditions differ from VLC in that they capture a much more specific type of variation, namely, conditional inclusion of code depending on the type of a class's type parameters; in contrast, VLC can represent arbitrary variation. Type conditions also have a quite coarse granularity, varying only top-level methods and fields. A feature, relative to VLC, is that different variants of the same code (class) can be used within the same program (by instantiating the class's type parameters differently).

When a new type system is designed or when new features are added to an existing system, a new unification algorithm and type inference algorithm must be coined for the new system, and the correctness of the new system and algorithms have to be demonstrated. As evidenced by this paper, this is quite a lot of work. To reduce this burden and promote reuse, Odersky and Sulzmann [1999; 2001; 2008] have proposed HM(X), a general framework for type systems with constraints, including a type inference algorithm that computes principal types that satisfy these constraints. By instantiating X to different extensions, different type systems can be generated from HM(X). For example, X can be instantiated to polymorphic records, equational theories, and subtypes. Variational type inference cannot be implemented within HM(X), however, and we cannot therefore reuse its algorithms and proofs. This is because HM(X) requires constraints to satisfy a regularity property that does not hold in variational type inference. The regularity property states that two sides of any equational theory must have the same free variables, but this is not true in VLC's type system because of choice domination. For example, $A\langle A\langle a,b\rangle,c\rangle \equiv A\langle a,c\rangle$ but $\{a,b,c\} \neq \{a,c\}$.

Some aspects of the type system presented in this paper can be simulated by dependent types [Xi and Pfenning 1999]. However, there are limitations of this approach. For one, type inference with dependent types is undecidable. Most dependent type systems also require programmers to supply complex type annotations or construct proof terms to support type checking [Paulin-Mohring 1993; Xi and Pfenning 1999; Fogarty et al. 2007; Norell 2007]. This is a significant burden that our type system does not impose. A more restricted version of choice types could also be implemented with phantom types and GADTs [Johann and Ghani 2008]. However, GADTs cannot express arbitrary choice types since the type of each alternative would be constrained by the requirement that the result type of each branch of a GADT must refine the data type being defined.

Related to our process of variation type normalization, Balat et al. [2004] present a powerful normalizer for terms in lambda calculus with sums. They make use of a similar transformation for eliminating dead alternatives. Our type normalizer differs from theirs in two technical details. First, choices in VLC are named and choices with different names are treated differently. Their normalizer makes no such distinction among sums, making it essentially equivalent to VLC in which all choices are in the same dimension. Second, the order of choice nesting is significant in our normalization, whereas the order of sum nesting is not in theirs.

Program variation can also be expressed using program generation or metaprogramming techniques. Using MetaML [Taha and Sheard 2000] one could represent variational code through the use of macros, which would be evaluated in one stage, leading to non-variational programs in the next. In this way one could simulate the variation annotations of VLC, and MetaML's static type system would ensure that all represented variations would be type correct. However, a serious limitation of that approach is that MetaML's type system would require that all alternatives in a choice macro have the same type. Template Haskell [Sheard and Jones 2002] is more flexible in this regard since it would allow alternatives of different types to be put into a choice macro. However, this flexibility is achieved by delaying type checking until after macro expansion, abandoning the static typing paradigm [Shields et al. 1998], and meaning that program variants would not be typed until after they are generated.

In C++, generic programming [Austern 1998] and template metaprogramming [Abrahams and Gurtovoy 2004] enable software variation through template instantiation. By instantiating templates with different arguments, different program variants can be generated. C++ templates provide a static type reflection mechanism [Garcia 2008] that allow metaprograms to query the type information of template parameters. At the same time, C++ templates support specialization to customize generic implementations for particular type parameters. These features allow C++ to achieve both maximum reusability and efficiency. However, since the complete type information of the template parameters is not available statically, full type checking is delayed until a particular program variant is generated [Stroustrup 1994]. This means it is not possible to statically type check all program variants without generating each one. This has led to serious usability problems [Gregor et al. 2006].

Despite huge efforts devoted to address this issue [Reis and Stroustrup 2005; Dos Reis and Stroustrup 2006; Gregor et al. 2006; Siek and Taha 2006; Garcia and Lumsdaine 2009; Miao and Siek 2010], no satisfying solution has been proposed. Järvi et al. [2006] conclude that to achieve modular type checking for C++ templates, either the use or implementation of specialization must be constricted. Choice types together with variational unification provide a possible mechanism for removing this constriction, by encoding and synchronizing the different possible types for template parameters.

SafeGen [Huang et al. 2005] and MorphJ [Huang and Smaragdakis 2008; 2011] provide a way to create generic Java classes whose methods are generated by iterating over the fields and methods of other classes. This is particularly suited for defining wrappers and proxies for existing classes. Method definition in MorphJ consists of two parts: (1) the iteration pattern that determines which methods will be matched, and the resulting method name, return type, and argument types, and (2) the method body, which may refer to pattern-matching vari-

ables defined in the iteration pattern. MorphJ checks that the operations applied to pattern-matching variables are supported by assumptions introduced in the iteration pattern. This is very similar to C++ *concepts*, which describe constraints on template parameters [Gregor et al. 2006]. Both approaches ensure that generated class will be well-typed if they are instantiated with arguments that satisfy the assumptions/constraints.

Many other techniques have been developed for ensuring the generation of well-formed and well-typed programs in mainstream object-oriented languages. Fändrich et al. [2006] propose a pattern-matching and template-based approach for writing reflective code in C#. Work on Maven [Goldman et al. 2010] and subsequent work [Disenfeld and Katz 2012] has addressed the problem of ensuring that specified behaviors are achieved after complex aspect-weaving operations. Finally, *expanders* [Warth et al. 2006] provide a new language construct for updating the methods and fields of an existing Java class in a non-invasive way. This work achieves modular type checking by statically scoping the usage of expanders.

10.3. Type Checking Software Product Lines

In the context of software product lines (SPLs), a lot of work has been done to improve the type checking of generated products and avoid the brute-force strategy of typing each product individually. One example is Thaker et al. [2007], who present an approach for type checking SPLs based on the safe composition of type-correct modules [Delaware et al. 2009]. This is given as a tool implemented in the AHEAD framework for feature-oriented software development [Batory et al. 2004], where each feature is implemented in a separate module. These modules can then be selectively composed into products, and the set of all such possible products forms a SPL. Safe composition of features is achieved in two steps. In the first, each module is compiled and checked to see whether it satisfies a lightweight global consistency property. After that, constraints between particular modules are checked. VLC does not consider these constraints separately, yet it supports both kinds of constraints. Another important difference between their approach and our own is that they represent variation at a much coarser level of granularity. The finest granularity of variation in their work is statements, while in VLC we support variation at arbitrarily fine-grained levels. Finally, their approach uses SAT solvers to ensure safe composition, whereas we infer types directly.

Other work on ensuring the type correctness of generated products within the compositional approach include the work of Apel and Hutchins [2008], which describes feature composition formally with a new calculus, and the work of Chae and Blume [2008], which ensures that the types of composed features will match.

Also in the field of SPLs, Kästner et al. [2012] describe a type system for Colored Featherweight Java (CFJ). In CFJ, parts of a Featherweight Java (FJ) program can be "colored", marking it as optional and associating it with a particular feature. They use the CIDE tool [Kästner et al. 2008] to enforce syntactic correctness of CFJ programs by allowing only the coloring of syntactically optional code. A variant is generated by selecting which features in the CFJ program to include. VLC and CFJ share many of the properties of annotative approaches—for example, both type systems have the property that variation selection preserves typing—but they differ in the kinds of variations that are supported. Specifically, VLC supports alternative variation, whereas CFJ only supports optional variation naturally and supports alternative variation only through external tools. On a conceptual level, CFJ is a combination of the type system for FJ and a path analysis that checks whether elements that are referred to are reachable. This leads to qualitatively different typing rules in CFJ, which are extensions of those for FJ with reachability checking and annotation propagation. In contrast, the type system for VLC is quite similar to other type systems. Also, like the work mentioned above, CFJ captures variation only at a statement level of granularity and uses a SAT solver to ensure type correctness, rather than inferring variational types.

While CFJ provides type checking support for annotative variation in Featherweight Java, Feature Featherweight Java (FFJ) [Apel et al. 2008] and FFJ_{PL} [Apel et al. 2010] provide

type checking support for compositional variation in the language. The goal of these languages is to explore the flexibility of class refinement and module composition and to ensure the type correctness of whole software product lines. Both FFJ and FFJ$_{PL}$ provide support for alternative types by supporting the definition of alternative classes; there are no constraints imposed on the implementations of alternative classes, or their relationships with other classes. There are two important differences between VLC and this line of work. First, choice types are associated with dimensions in VLC, allowing synchronization between different variation points at the type-level, which cannot be expressed in FFJ or FFJ$_{PL}$. Second, choice types can be simplified (e.g. through choice idempotency), allowing for simpler variation at the type level than at the expression level.

There has also been some work on statically checking variational C programs containing CPP annotations. Initial work in this area was done by Aversano et al. [2002], where they demonstrate the widespread use of conditionally declared variables with potentially different types and the difficulty in ensuring that they are used correctly in all variants. As a solution, they propose the construction of an extended symbol table with the conditions in which each symbol is defined and has the corresponding type. Kenner et al. [2010] provide a working implementation of essentially this approach in TypeChef, although it currently ensures only that symbol references are satisfied in all variants and that no symbols are redefined. TypeChef's ultimate goal is to be able to efficiently ensure the type correctness of all variants of CPP-annotated C programs, which becomes promising with the work of variability-aware parsing [Kästner et al. 2011; Gazzillo and Grimm 2012]. There is a huge amount of engineering overhead in such a project, not related to variational type systems, because of CPP's somewhat quirky semantics and highly unstructured variation representation. For example, making a selection in VLC roughly corresponds to setting a macro in CPP, but a macro's setting can change several times throughout a single run of the C preprocessor, making it much more difficult to even determine which code corresponds to a particular variant. Therefore, this work is mostly complementary to our own; we abstract these challenges away by focusing on a simple formal language (VLC), allowing us to focus on the core issue of typing variational programs. Also, like their previous work on CFJ, TypeChef is constraint-based and relies on a constraint solver for checking properties, while we can more generally infer types.

In this paper a VLC expression is typable only if all of its variants are well typed. Thus a type error in one variant causes the entire variational program to be type incorrect. Elsewhere we have shown how to extend the approach described in this paper to be error-tolerant [2012]. The extension introduces *partial* variational types that may also contain errors for some variants, which are introduced when unification fails. This extension is useful not only for locating type errors, but also for supporting the incremental development of variational software. The results and techniques related to this extension are mostly orthogonal to the results presented here.

Aside from type checking, many other static analyses on variational software have been investigated. Through the composition of proofs for individual feature modules, Delaware et al. [2011] provide a way to derive proofs for individual products. Other work includes variability-aware dataflow analysis [Brabrand et al. 2012; Liebig et al. 2012], and variability-aware model checking [Classen et al. 2010; Classen et al. 2011; Cordy et al. 2012; Apel et al. 2013].

11. CONCLUSIONS AND FUTURE WORK

We have presented a method to infer types for variational programs. Our solution addresses both the issues of *efficiently* ensuring the type correctness of all program variants, and effectively *representing* variational types. The contributions of this work are as follows:

1. The *VLC language*, a simple formal language that supports theoretical research on variational software. Research in this area has so far been mostly tool-based, but the success of lambda calculus in the programming languages community demonstrates the utility of such a research tool.
2. The notion of *variational types*, which solve the problem of effectively representing types in variational programs. This not only directly supports variational type inference, but can also support the understanding of variational programs.
3. A *type system* for VLC that maps VLC expressions onto variational types. The type system relies on an equivalence relationship between types, which has a corresponding confluent and normalizing rewrite relation that facilitates the checking of type equivalence. As a fundamental result we have shown that types are preserved across variation elimination, which is an important precondition for the correctness of the integrated type checking of variational software. In addition, we have demonstrated how this type system can be extended to support new typing features, in order to support the application of variational typing to real programming languages.
4. A *type inference algorithm* that infers variational types for VLC expressions, expressed as a straightforward extension of the well-known algorithm $\mathcal{W}$.
5. A *unification algorithm* for variational types, which is the most significant component of the type inference algorithm. Part of our solution to this problem is the concept of qualified type variables that allows the assignment of different types to the same type variable when it occurs in different variational branches of a type.
6. A demonstration that although the unification problem is equational and contains distributivity and associativity laws, it is *decidable* and *unitary*, because we have added type dominance as an additional equivalence relationship.
7. A *complexity analysis* of the unification algorithm and a characterization of the efficiency gains offered by variational type inference over typing individual variants.
8. A demonstration that important properties from other lambda calculus type systems are preserved in the type system for VLC. For example, that the type inference algorithm is *sound*, *complete*, and has the *principal type property*. That these properties can be preserved in a variational extension of lambda calculus is encouraging for the addition of variational types to more sophisticated type systems.

In future work, we will consider applying the choice calculus and variational unification to other analyses. The ability to encapsulate different types in choice types and unify them with our unification algorithm suggests several potential applications.

One possibility is increasing the type safety guarantees in metaprogramming. The challenge of a type system for an expressive metalanguage is how to represent and reason about uncertain types for expressions. By capturing all potential types of an expression as alternatives in a choice type, we can eliminate this uncertainty. We are currently investigating a type system for C++ template metaprogramming using this approach. Relatedly, we can use this approach to infer constraints on metaprograms (such as C++ concepts [Gregor et al. 2006] and MorphJ assumption specifications [Huang and Smaragdakis 2008]), rather than requiring users to write these constraints explicitly.

Another possible application is lifting existing unification-based pointer analyses [Steensgaard 1996; Das 2000] to variational programs, allowing pointer analyses on whole SPLs.

ACKNOWLEDGMENTS

We would like to thank Christian Kästner and the anonymous reviewers for many helpful comments on earlier drafts of this paper. This work is supported by the Air Force Office of Scientific Research under the grant FA9550-09-1-0229 and by the National Science Foundation under the grants CCF-0917092 and and CCF-1219165.

REFERENCES

ABRAHAMS, D. AND GURTOVOY, A. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional.

ANANTHARAMAN, S., NARENDRAN, P., AND RUSINOWITCH, M. 2004. Unification Modulo ACUI Plus Homomorphisms/Distributivity. *Journal of Automated Reasoning 33*, 1–28.

APEL, S. AND HUTCHINS, D. 2008. A Calculus for Uniform Feature Composition. *ACM Trans. Program. Lang. Syst. 32,* 5, 19:1–19:33.

APEL, S. AND KÄSTNER, C. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology 8,* 5, 49–84.

APEL, S., KÄSTNER, C., GRÖßLINGER, A., AND LENGAUER, C. 2010. Type Safety for Feature-Oriented Product Lines. *Automated Software Engg. 17,* 3, 251–300.

APEL, S., KÄSTNER, C., AND LENGAUER, C. 2008. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Int. Conf. on Generative Programming and Component Engineering*. 101–112.

APEL, S., RHEIN, A. V., WENDLER, P., GRÖSSLINGER, A., AND BEYER, D. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. In *IEEE Int. Conf. on Software Engineering*. IEEE. To appear.

AUSTERN, M. H. 1998. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman.

AVERSANO, L., PENTA, M. D., AND BAXTER, I. D. 2002. Handling Preprocessor-Conditioned Declarations. In *Int. Workshop on Source Code Analysis and Manipulation*. 83–92.

BAADER, F. AND NIPKOW, T. 1998. *Term Rewriting and All That*. Cambridge University Press.

BAADER, F. AND SNYDER, W. 2001. Unification Theory. In *Handbook of Automated Reasoning*, A. Robinson and A. Voronkov, Eds. Vol. I. North Holland, Chapter 8, 445–532.

BALAT, V., COSMO, R. D., AND FIORE, M. P. 2004. Extensional Normalisation and Type-Directed Partial Evaluation for Typed Lambda Calculus with Sums. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 64–76.

BATORY, D., SARVELA, J. N., AND RAUSCHMAYER, A. 2004. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering 30,* 6, 355–371.

BRABRAND, C., RIBEIRO, M., TOLÊDO, T., AND BORBA, P. 2012. Intraprocedural Dataflow Analysis for Software Product Lines. In *Int. Conf. on Aspect-Oriented Software Development*. AOSD '12. ACM, New York, NY, USA, 13–24.

BRACHA, G. AND COOK, W. 1990. Mixin-Based Inheritance. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. 303–311.

CHAE, W. AND BLUME, M. 2008. Building a Family of Compilers. In *Int. Software Product Line Conf.* SPLC '08. IEEE Computer Society, Washington, DC, USA, 307–316.

CHEN, S., ERWIG, M., AND WALKINGSHAW, E. 2012. An error-tolerant type system for variational lambda calculus. In *the 17th ACM SIGPLAN International Conference on Functional Programming*. 29–40.

CLASSEN, A., HEYMANS, P., SCHOBBENS, P.-Y., AND LEGAY, A. 2011. Symbolic Model Checking of Software Product Lines. In *IEEE Int. Conf. on Software Engineering*. ICSE '11. ACM, New York, NY, USA, 321–330.

CLASSEN, A., HEYMANS, P., SCHOBBENS, P.-Y., LEGAY, A., AND RASKIN, J.-F. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*. ICSE '10. ACM, New York, NY, USA, 335–344.

CORDY, M., CLASSEN, A., PERROUIN, G., SCHOBBENS, P.-Y., HEYMANS, P., AND LEGAY, A. 2012. Simulation-based Abstractions for Software Product-Line Model Checking. In *IEEE Int. Conf. on Software Engineering*. ICSE 2012. IEEE Press, Piscataway, NJ, USA, 672–682.

CZARNECKI, K. AND EISENECKER, U. W. 2000. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

DAMAS, L. AND MILNER, R. 1982. Principal Type Schemes for Functional Programming Languages. In *9th ACM Symp. on Principles of Programming Languages*. 207–208.

DAS, M. 2000. Unification-based Pointer Analysis with Directional Assignments. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*. PLDI '00. ACM, New York, NY, USA, 35–46.

DELAWARE, B., COOK, W., AND BATORY, D. 2009. A Machine-Checked Model of Safe Composition. In *Workshop on Foundations of Aspect-Oriented Languages*. 31–35.

DELAWARE, B., COOK, W., AND BATORY, D. 2011. Product lines of theorems. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '11. ACM, New York, NY, USA, 595–608.

DEZANI-CIANCAGLINI, M., GHILEZAN, S., AND VENNERI, B. 1997. The "Relevance" of Intersection and Union Types. *Notre Dame Journal of Formal Logic 38,* 2, 246–269.

DISENFELD, C. AND KATZ, S. 2012. A Closer Look at Aspect Interference and Cooperation. In *Int. Conf. on Aspect-Oriented Software Development*. AOSD '12. ACM, New York, NY, USA, 107–118.

DOS REIS, G. AND STROUSTRUP, B. 2006. Specifying C++ Concepts. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. 295–308.

DOWNEY, P. J., SETHI, R., AND TARJAN, R. E. 1980. Variations on the Common Subexpression Problem. *J. ACM 27,* 4, 758–771.

ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-Oriented Programming. *Comm. of the ACM 44,* 10, 28–32.

ERWIG, M. AND WALKINGSHAW, E. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Transactions on Software Engineering and Methodology 21,* 1, 6:1–6:27.

FÄHNDRICH, M., CARBIN, M., AND LARUS, J. R. 2006. Reflective Program Generation with Patterns. In *Int. Conf. on Generative Programming and Component Engineering*. GPCE '06. ACM, New York, NY, USA, 275–284.

FLATT, M. AND PLT. 2010. Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. `http://racket-lang.org/tr1/`.

FOGARTY, S., PASALIC, E., SIEK, J., AND TAHA, W. 2007. Concoqtion: Indexed Types Now! In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '07. ACM, New York, NY, USA, 112–121.

GARCIA, R. 2008. Static Computation and Reflection. Ph.D. thesis, Indiana University.

GARCIA, R. AND LUMSDAINE, A. 2009. Toward Foundations for Type-Reflective Metaprogramming. In *Int. Conf. on Generative Programming and Component Engineering*. GPCE '09. 25–34.

GAZZILLO, P. AND GRIMM, R. 2012. SuperC: Parsing all of C by Taming the Preprocessor. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*. PLDI '12. ACM, New York, NY, USA, 323–334.

GNU PROJECT. 2009. *The C Preprocessor*. Free Software Foundation. `http://gcc.gnu.org/onlinedocs/cpp/`.

GOLDMAN, M., KATZ, E., AND KATZ, S. 2010. Maven: Modular Aspect Verification and Interference Analysis. *Formal Methods in System Design 37,* 1, 61–92.

GREGOR, D., JÄRVI, J., SIEK, J., STROUSTRUP, B., DOS REIS, G., AND LUMSDAINE, A. 2006. Concepts: Linguistic Support for Generic Programming in C++. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '06. 291–310.

HUANG, S. S. AND SMARAGDAKIS, Y. 2008. Expressive and Safe Static Reflection with MorphJ. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*. PLDI '08. ACM, New York, NY, USA, 79–89.

HUANG, S. S. AND SMARAGDAKIS, Y. 2011. Morphing: Structurally Shaping a Class by Reflecting on Others. *ACM Trans. Program. Lang. Syst. 33,* 2, 6:1–6:44.

HUANG, S. S., ZOOK, D., AND SMARAGDAKIS, Y. 2005. Statically Safe Program Generation with Safegen. In *Int. Conf. on Generative Programming and Component Engineering*. GPCE'05. Springer-Verlag, Berlin, Heidelberg, 309–326.

HUANG, S. S., ZOOK, D., AND SMARAGDAKIS, Y. 2007. cJ: Enhancing Java with Safe Type Conditions. In *Int. Conf. on Aspect-Oriented Software Development*. 185–198.

JÄRVI, J., GREGOR, D., WILLCOCK, J., LUMSDAINE, A., AND SIEK, J. 2006. Algorithm Specialization in Generic Programming: Challenges of Constrained Generics in C++. In *ACM SIGPLAN Conf. on Programming language Design and Implementation*. PLDI '06. ACM, New York, NY, USA, 272–282.

JOHANN, P. AND GHANI, N. 2008. Foundations for Structured Programming with GADTs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. ACM, New York, NY, USA, 297–308.

KAGAWA, K. 2006. Polymorphic Variants in Haskell. In *ACM SIGPLAN Workshop on Haskell*. ACM, 37–47.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University. Nov.

KÄSTNER, C., APEL, S., AND KUHLEMANN, M. 2008. Granularity in Software Product Lines. In *IEEE Int. Conf. on Software Engineering*. 311–320.

KÄSTNER, C., APEL, S., THÜM, T., AND SAAKE, G. 2012. Type checking annotation-based product lines. *ACM Transactions on Software Engineering and Methodology 21,* 3, 14:1–14:39.

KÄSTNER, C., GIARRUSSO, P. G., RENDEL, T., ERDWEG, S., OSTERMANN, K., AND BERGER, T. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, 805–824.

KENNER, A., KÄSTNER, C., HAASE, S., AND LEICH, T. 2010. TypeChef: Toward Type Checking #ifdef Variability in C. In *Int. Workshop on Feature-Oriented Software Development*. 25–32.

KIM, C. H. P., KÄSTNER, C., AND BATORY, D. 2008. On the Modularity of Feature Interactions. In *Int. Conf. on Generative Programming and Component Engineering*. 19–23.

LIEBIG, J., VON RHEIN, A., KÄSTNER, C., APEL, S., DÖRRE, J., AND LENGAUER, C. 2012. Large-Scale Variability-Aware Type Checking and Dataflow Analysis. Number MIP-1212.

MEZINI, M. AND OSTERMANN, K. 2003. Conquering Aspects with Caesar. In *Int. Conf. on Aspect-Oriented Software Development*. ACM, New York, NY, USA, 90–99.

MEZINI, M. AND OSTERMANN, K. 2004. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes 29,* 6, 127–136.

MIAO, W. AND SIEK, J. G. 2010. Incremental Type-checking for Type-Reflective Metaprograms. In *Int. Conf. on Generative Programming and Component Engineering*. GPCE '10. 167–176.

NELSON, G. AND OPPEN, D. C. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM 27,* 2, 356–364.

NORELL, U. 2007. Towards a Practical Programming Language Based on Dependent Type Theory. Ph.D. thesis, Department of Computer Sceince and Engineering, Chalmers University of Technology and Göteborg University.

ODERSKY, M., SULZMANN, M., AND WEHR, M. 1999. Type Inference with Constrained Types. *Theory and Practice of Object Systems 5,* 1, 35–55.

PAULIN-MOHRING, C. 1993. Inductive Definitions in the System Coq Rules and Properties. *Typed Lambda Calculi and Applications*, 328–345.

PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

POHL, K., BÖCKLE, G., AND VAN DER LINDEN, F. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg.

REIS, G. D. AND STROUSTRUP, B. 2005. A Formalism for C++. Tech. rep., ISO/IEC SC22/JTC1/WG21. October.

ROBINSON, J. A. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM 12,* 1, 23–41.

SHEARD, T. AND JONES, S. P. 2002. Template Meta-programming for Haskell. *SIGPLAN Not. 37,* 12, 60–75.

SHIELDS, M., SHEARD, T., AND PEYTON JONES, S. 1998. Dynamic Typing as Staged Type Inference. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '98. 289–302.

SIEK, J. AND TAHA, W. 2006. A Semantic Analysis of C++ Templates. In *European Conf. on Object-Oriented Programming*. ECOOP'06. Springer-Verlag, Berlin, Heidelberg, 304–327.

STEENSGAARD, B. 1996. Points-to Analysis in Almost Linear Time. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '96. ACM, New York, NY, USA, 32–41.

STROUSTRUP, B. 1994. *The Design and Evolution of C++*. ACM Press.

SULZMANN, M. 2001. A General Type Inference Framework for Hindley/Milner Style Systems. In *Int. Symp. on Functional and Logic Programming*. Springer-Verlag, 248–263.

SULZMANN, M. AND STUCKEY, P. J. 2008. HM(X) Type Inference is CLP(X) Solving. *J. Funct. Program. 18,* 2, 251–283.

TAHA, W. AND SHEARD, T. 2000. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science 248,* 1–2, 211–242.

THAKER, S., BATORY, D., KITCHIN, D., AND COOK, W. 2007. Safe Composition of Product Lines. In *Int. Conf. on Generative Programming and Component Engineering*. 95–104.

WARTH, A., STANOJEVIĆ, M., AND MILLSTEIN, T. 2006. Statically Scoped Object Adaptation with Expanders. In *ACM SIGPLAN Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '06. ACM, New York, NY, USA, 37–56.

XI, H. AND PFENNING, F. 1999. Dependent Types in Practical Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '99. ACM, New York, NY, USA, 214–227.

A. COLLECTED PROOFS

This appendix contains proofs that were too long to include in the body of the paper. Section A.1 provides a proof that the CT-unification problem is unitary, captured by Theorem 7.1 and discussed in Section 7.1. The proof contains several intermediate results. Section A.2 includes proofs for intermediate results related to the correctness of the unification algorithm, captured in Theorem 7.6 and discussed in Section 7.3. Finally, Section A.3 collects several other proofs from throughout the paper. In each case we first repeat the result being proved, for reference.

A.1. Proof that the CT-Unification Problem is Unitary

THEOREM 7.1. *Given a CT-unification problem U, there is a unifier σ such that for any unifier σ', there exists a mapping θ such that $\sigma' = \theta \circ \sigma$.*

In the following, we use U and Q to denote unification problems. We use T_L and T_R to denote the LHS and RHS of U, and T'_L and T'_R to denote the LHS and RHS of Q. We use $vars(U)$ to refer to all of the type variables in U. We also extend the notion of selection to unification problems and mappings by propagating the selection along to the types they contain, as defined below.

$$\lfloor T_L \equiv^? T_R \rfloor_s = \lfloor T_L \rfloor_s \equiv^? \lfloor T_R \rfloor_s$$
$$\lfloor \sigma \rfloor_s = \{(a, \lfloor T \rfloor_s) \mid (a, T) \in \sigma\}$$

The following lemma states that selection further extends over type substitution in a homomorphic way.

LEMMA A.1 (SELECTION EXTENDS OVER SUBSTITUTION). $\lfloor T\sigma \rfloor_s = \lfloor T \rfloor_s \lfloor \sigma \rfloor_s$

PROOF OF LEMMA A.1. The proof is based on induction over the structure of T and σ. We show the proof only for the most interesting cases where T is a choice type, and where T is a type variable mapped to a choice type in σ.

(1) Given $T = D\langle T_1, T_2\rangle$, assume $s = \tilde{D}$ (the case for $s = D$ is dual).

$$\begin{aligned}
\lfloor T\sigma \rfloor_s &= \lfloor D\langle T_1, T_2\rangle\sigma \rfloor_{\tilde{D}} && \text{by definition}\\
&= \lfloor D\langle T_1\sigma, T_2\sigma\rangle \rfloor_{\tilde{D}} && \text{type substitution}\\
&= \lfloor T_2\sigma \rfloor_{\tilde{D}} && \text{selection}\\
&= \lfloor T_2 \rfloor_{\tilde{D}} \lfloor \sigma \rfloor_{\tilde{D}} && \text{induction hypopethesis}\\
&= \lfloor D\langle T_1, T_2\rangle \rfloor_{\tilde{D}} \lfloor \sigma \rfloor_{\tilde{D}} && \text{selection}\\
&= \lfloor T \rfloor_s \lfloor \sigma \rfloor_s
\end{aligned}$$

(2) Given $T = a$, assume $a\sigma = D\langle T_1, T_2\rangle$ and $s = \tilde{D}$ (again $s = D$ is dual).

$$\begin{aligned}
\lfloor T \rfloor_s \lfloor \sigma \rfloor_s &= \lfloor a \rfloor_{\tilde{D}} \lfloor \sigma \rfloor_{\tilde{D}} && \text{by definition}\\
&= a\sigma'[a = \lfloor T_2 \rfloor_{\tilde{D}}] && \text{selection}\\
&= \lfloor T_2 \rfloor_{\tilde{D}} &&\\
\lfloor T\sigma \rfloor_s &= \lfloor a\sigma \rfloor_{\tilde{D}} && \text{by definition}\\
&= \lfloor D\langle T_1, T_2\rangle \rfloor_{\tilde{D}} && \text{by assumption}\\
&= \lfloor T_2 \rfloor_{\tilde{D}} = \lfloor T \rfloor_s \lfloor \sigma \rfloor_s &&
\end{aligned}$$

The remaining cases can be constructed similarly. □

From Lemma A.1, it follows by induction that the same result holds for decisions as for single selectors: $\lfloor T\sigma \rfloor_\delta = \lfloor T \rfloor_\delta \lfloor \sigma \rfloor_\delta$. Combining this with Lemma 4.2 (selection preserves type equivalency), we see that if σ is a unifier for U then $\lfloor T_L\sigma \rfloor_\delta \equiv \lfloor T_R\sigma \rfloor_\delta$ for any δ. This is the same as saying that if $T_L\sigma \not\equiv T_R\sigma$, then $\exists \delta : \lfloor T_L\sigma \rfloor_\delta \not\equiv \lfloor T_R\sigma \rfloor_\delta$. A direct consequence of this result is that if δ is super-complete (it eliminates all choice types in T_L, T_R, and σ) and $\lfloor T_L \rfloor_\delta \lfloor \sigma \rfloor_\delta \equiv \lfloor T_R \rfloor_\delta \lfloor \sigma \rfloor_\delta$, then σ is a unifier for U.

PROOF OF THEOREM 7.1. Using the type splitting algorithm described in Section 7.2, we can transform U into Q such that for all super-complete decisions $\delta_1, \delta_2, \cdots, \delta_n$, if $\delta_i \neq \delta_j$, then $vars(\lfloor Q \rfloor_{\delta_i}) \cap vars(\lfloor Q \rfloor_{\delta_j}) = \varnothing$.

Each subproblem $\lfloor Q \rfloor_{\delta_i}$ corresponding to a super-complete decision δ_i is plain. Therefore, we can obtain (via Robinson's algorithm) an mgu σ_i such that $\lfloor T'_L\sigma_i \rfloor_{\delta_i} \equiv \lfloor T'_R\sigma_i \rfloor_{\delta_i}$. Let σ be the disjoint union of all of these mgus, $\sigma = \bigcup_{i \in \{1..n\}} \sigma_i$.

Since the type variables in each subproblem are different, for each subproblem we have $\lfloor T'_L\sigma \rfloor_{\delta_i} \equiv \lfloor T'_R\sigma \rfloor_{\delta_i}$. Then based on the discussion after Lemma A.1, σ is a unifier for $T'_L \equiv^?_q T'_R$. Moreover, it is most general by construction since each σ_i is most general. Based on Theorem 7.9 (variational unification is sound) and Lemma 7.7 (*comp* is correct and preserves principality), the completion of σ is the mgu for U, which proves that variational unification is unitary. □

A.2. Proofs that the Variational Unification Algorithm is Correct

This section contains several proofs related to the correctness of the unification algorithm. The corresponding theorems appear in Section 7.3.

A.2.1. Proof of Theorem 7.5

THEOREM 7.5 (SOUNDNESS). *If* $unify(T_1, T_2) = \sigma$, *then* $T_1\sigma \equiv T_2\sigma$.

PROOF. The proof is by induction on the structure of T_1 and T_2. To make the proof easier to follow, we do this by stepping through each case of the *unify* algorithm, briefly describing why the theorem holds for each base case, or why it is preserved for recursive cases. For many cases, correctness is preserved by *unify* being recursively invoked on semantically equivalent arguments.

1. Both types are plain. The result is determined by the Robinson unification algorithm, which is known to be correct [Robinson 1965].
2. A qualified type variable a_q and a choice type. Correctness is preserved since $a_q \equiv D\langle a_{Dq}, a_{\bar{D}q}\rangle$.
3. Two choice types in the same dimension. Decomposition by alternatives is correct by the inductive hypothesis and Lemma 7.2.
4. The next three cases consider choice types in different dimensions. They preserve correctness for the following reasons.
   (a) Hoisting is semantics preserving.
   (b) Splitting is variable independent by Lemma 7.3.
   (c) Splitting is choice independent by Lemma 7.4.
5. The next two cases consider unifying a choice type with a non-choice type. Correctness is preserved in both cases since the recursive calls are on semantically equivalent arguments, by choice idempotency.
6. Two function types. Given the inductive hypotheses, $T_1\sigma \equiv T_1'\sigma$ and $T_2\sigma \equiv T_2'\sigma$, we can construct $(T_1 \to T_2)\sigma \equiv (T_1' \to T_2')\sigma$ by an application of the FUN equivalence rule in Figure 5.
7. The last case considers a qualified type variable a_q and a function type $T \to T'$. If the occurs check fails, the theorem is trivially satisfied since the condition of the implication is not met. If it succeeds, the theorem is satisfied by the definition of substitution. □

A.2.2. Proof of Lemma 7.7

LEMMA 7.7. *Given a mapping* $\{a_q \mapsto T'\}$, *if* $T = comp(q, T', b)$ *is the completed type (where* b *is fresh), then* $\lfloor T \rfloor_q = \lfloor T' \rfloor_q$. *More generally, given* $\{a_{q_1} \mapsto T_1, \ldots, a_{q_n} \mapsto T_n\}$, *if* $T = comp(q_1, T_1, comp(q_2, T_2, \ldots comp(q_n, T_n, b)\ldots))$, *then for every* $q_i \in \{q_1 \ldots q_n\}$, *we have* $\lfloor T \rfloor_{q_i} = \lfloor T_i \rfloor_{q_i}$.

PROOF. We can prove the first part of this theorem by structural induction on the qualifier q. The base case, where q is the empty qualifier ϵ, is trivial since $comp(\epsilon, T', b) = T'$. We show the inductive case below for $q = Dq'$ (the case for $q = \bar{D}q'$ is a dual). Note that the induction hypothesis is $\lfloor comp(q', T', b) \rfloor_{q'} = \lfloor T' \rfloor_{q'}$.

$$
\begin{aligned}
\lfloor T \rfloor_q &= \lfloor comp(Dq', T', b) \rfloor_{Dq'} && \text{by assumption} \\
&= \lfloor D\langle comp(q', T', b), fresh(b)\rangle \rfloor_{Dq'} && \text{definition of } comp \\
&= \lfloor \lfloor comp(q', T', b) \rfloor_D \rfloor_{q'} && \text{definition of repeated selection} \\
&= \lfloor \lfloor comp(q', T', b) \rfloor_{q'} \rfloor_D && \text{selector ordering is irrelevant} \\
&= \lfloor \lfloor T' \rfloor_{q'} \rfloor_D && \text{induction hypothesis} \\
&= \lfloor T' \rfloor_q && \text{selector ordering is irrelevant}
\end{aligned}
$$

We can prove the second part by induction on the mapping of qualified type variables, using the result from the first part and the observation that *comp* is commutative, for example, $comp(q_1, T_1, comp(q_2, T_2, b)) \equiv comp(q_2, T_2, comp(q_1, T_1, b))$. □

A.2.3. Proof sketch of Lemma 7.8

LEMMA 7.8 (COMPLETION). *Given a CT-unification problem* $T_L \equiv^? T_R$ *and the corresponding qualified unification problem* $T_L' \equiv_q^? T_R'$, *if* σ_Q *is a unifier for* $T_L' \equiv_q^? T_R'$ *and* σ_U *is the unifier attained by completing* σ_Q, *then for any super-complete decision* δ, $\lfloor T_L\sigma_U \rfloor_\delta \equiv \lfloor T_L'\sigma_Q \rfloor_\delta$ *and* $\lfloor T_R\sigma_U \rfloor_\delta \equiv \lfloor T_R'\sigma_Q \rfloor_\delta$.

PROOF SKETCH. The proof is based on induction over the structure of the types T_L and T_R, the super-complete decision δ, and the unifier σ_Q. We show the proof for several cases; the remaining cases can be derived similarly.

(1) $T'_L = T_L$, that is, T_L is a plain type. There are many sub-cases; we show two of them:

(a) $T_L = a$, a type variable. If $(a, T_a) \in \sigma_Q$, then by the definition of completion, $(a, T_a) \in \sigma_U$. Thus, $\lfloor T_L \sigma_U \rfloor_\delta = \lfloor T_a \rfloor_\delta = \lfloor T'_L \sigma_Q \rfloor_\delta$.

(b) $T_L = T_{arg} \to T_{res}$. The induction hypothesis is that $\lfloor T_{arg}\sigma_U \rfloor_\delta = \lfloor T_{arg}\sigma_Q \rfloor_\delta$ and $\lfloor T_{res}\sigma_U \rfloor_\delta = \lfloor T_{res}\sigma_Q \rfloor_\delta$. Then $\lfloor (T_{arg} \to T_{res})\sigma_U \rfloor_\delta = \lfloor T_{arg}\sigma_U \rfloor_\delta \to \lfloor T_{res}\sigma_U \rfloor_\delta = \cdots = \lfloor (T_{arg} \to T_{res})\sigma_Q \rfloor_\delta$.

(2) $T_L = D\langle T_1, T_2 \rangle$, $\delta = \tilde{D}\delta_1$. Let $T'_L = D\langle T'_1, T'_2 \rangle$, the induction hypothesis is that $\lfloor T_1\sigma_U \rfloor_{\delta_1} = \lfloor T'_1\sigma_Q \rfloor_{\delta_1}$ and $\lfloor T_2\sigma_U \rfloor_{\delta_1} = \lfloor T'_2\sigma_Q \rfloor_{\delta_1}$ for any δ_1.

$$
\begin{aligned}
\lfloor T_L\sigma_U \rfloor_\delta &= \lfloor D\langle T_1, T_2\rangle \sigma_U \rfloor_{\tilde{D}\delta_1} && \text{by assumption} \\
&= \lfloor \lfloor T_2 \rfloor_{\tilde{D}} \lfloor \sigma_U \rfloor_{\tilde{D}} \rfloor_{\delta_1} && \text{selection} \\
&= \lfloor \lfloor T_2\sigma_U \rfloor_{\tilde{D}} \rfloor_{\delta_1} && \text{by Lemma A.1} \\
&= \lfloor \lfloor T_2\sigma_U \rfloor_{\delta_1} \rfloor_{\tilde{D}} && \text{selector ordering is irrelevant} \\
&= \lfloor \lfloor T'_2\sigma_Q \rfloor_{\delta_1} \rfloor_{\tilde{D}} && \text{induction hypothesis} \\
\lfloor T'_L\sigma_Q \rfloor_\delta &= \ldots = \lfloor \lfloor T'_2\sigma_Q \rfloor_{\delta_1} \rfloor_{\tilde{D}}
\end{aligned}
$$

(3) $T_L = D\langle a, T_2 \rangle$, $\delta = D$. In this case, $T'_L = D\langle a_D, T'_2 \rangle$.

$$
\begin{aligned}
\lfloor T_L\sigma_U \rfloor_\delta &= \lfloor D\langle a, T_2\rangle \sigma_U \rfloor_D && \text{by assumption} \\
&= \lfloor a\sigma_U \rfloor_D && \text{selection and Lemma A.1} \\
&= \lfloor a_D\sigma_Q \rfloor_D && \text{by Lemma 7.7} \\
&= \lfloor D\langle a_D, T'_2\rangle \sigma_Q \rfloor_D && \text{selection} \\
&= \lfloor T'_L\sigma_Q \rfloor_\delta
\end{aligned}
$$

The cases for T_R are dual. □

A.3. Other Collected Proofs

A.3.1. Proof sketch of Lemma 4.2

LEMMA 4.2 (TYPE EQUIVALENCE PRESERVATION). *If $T_1 \equiv T_2$, then $\lfloor T_1 \rfloor_s \equiv \lfloor T_2 \rfloor_s$.*

PROOF SKETCH. The proof of this lemma proceeds by case, demonstrating that for each equivalence rule defined in Figure 5, if we apply the same selector s to both the LHS and the RHS of the rule, the resulting expressions are still equivalent. We demonstrate this for only a few cases, but the other cases can be treated similarly.

First, we consider the F-C rule. There are two sub-cases to consider: either the dimension of the choice type matches that of the selector, or it does not. We consider the sub-case where the dimension name does not match first.

$$
\begin{aligned}
\lfloor D\langle T_1 \to T'_1, T_2 \to T'_2\rangle \rfloor_s &= D\langle \lfloor T_1 \rfloor_s \to \lfloor T'_1 \rfloor_s, \lfloor T_2 \rfloor_s \to \lfloor T'_2 \rfloor_s \rangle && \text{selection in LHS} \\
&\equiv D\langle \lfloor T_1 \rfloor_s, \lfloor T_2 \rfloor_s \rangle \to D\langle \lfloor T'_1 \rfloor_s, \lfloor T'_2 \rfloor_s \rangle && \text{by the rule F-C} \\
\lfloor D\langle T_1, T_2\rangle \to D\langle T'_1, T'_2\rangle \rfloor_s &= \lfloor D\langle T_1, T_2\rangle \rfloor_s \to \lfloor D\langle T'_1, T'_2\rangle \rfloor_s && \text{selection in RHS} \\
&= D\langle \lfloor T_1 \rfloor_s, \lfloor T_2 \rfloor_s \rangle \to D\langle \lfloor T'_1 \rfloor_s, \lfloor T'_2 \rfloor_s \rangle && \text{by definition}
\end{aligned}
$$

For the sub-case where the dimension name matches, there are two further sub-cases, depending on whether we are selecting the first or second alternatives in dimension D. Below we show the case for $s = \tilde{D}$ (selecting the second alternatives). The case for $s = D$ is dual to this.

$$
\begin{aligned}
\lfloor D\langle T_1 \to T'_1, T_2 \to T'_2\rangle \rfloor_{\tilde{D}} &= \lfloor T_2 \to T'_2 \rfloor_{\tilde{D}} && \text{selection in LHS} \\
&= \lfloor T_2 \rfloor_{\tilde{D}} \to \lfloor T_2 \rfloor_{\tilde{D}} && \text{by definition} \\
\lfloor D\langle T_1, T_2\rangle \to D\langle T'_1, T'_2\rangle \rfloor_{\tilde{D}} &= \lfloor D\langle T_1, T_2\rangle \rfloor_{\tilde{D}} \to \lfloor D\langle T'_1, T'_2\rangle \rfloor_{\tilde{D}} && \text{selection in RHS} \\
&= \lfloor T_2 \rfloor_{\tilde{D}} \to \lfloor T'_2 \rfloor_{\tilde{D}} && \text{by definition}
\end{aligned}
$$

Next we consider the C-C-SWAP2 rule. Here there are three cases to consider: s makes a selection in dimension D, in dimension D', or in some other dimension. The case for when s makes a selection in D follows.

$$\begin{aligned}
\lfloor D'\langle T_1, D\langle T_2, T_3\rangle\rangle\rfloor_{\tilde{D}} &= D'\langle \lfloor T_1\rfloor_{\tilde{D}}, \lfloor D\langle T_2, T_3\rangle\rfloor_{\tilde{D}}\rangle && \text{selection in LHS}\\
&= D'\langle \lfloor T_1\rfloor_{\tilde{D}}, \lfloor T_3\rfloor_{\tilde{D}}\rangle && \text{by definition}\\
\lfloor D\langle D'\langle T_1, T_2\rangle, D'\langle T_1, T_3\rangle\rangle\rfloor_{\tilde{D}} &= \lfloor D'\langle T_1, T_3\rangle\rfloor_{\tilde{D}} && \text{selection in RHS}\\
&= D'\langle \lfloor T_1\rfloor_{\tilde{D}}, \lfloor T_3\rfloor_{\tilde{D}}\rangle && \text{by definition}
\end{aligned}$$

The second case is a dual to this, and the third case can be proved in a similar way as the first case for the F-C rule. The proofs of the remaining rules proceed in a similar fashion. □

A.3.2. Proof of Lemma 4.4

LEMMA 4.4 (LOCAL CONFLUENCE). *For any type T, if $T \rightsquigarrow T_1$ and $T \rightsquigarrow T_2$, then there exists some type T' such that $T_1 \rightsquigarrow^* T'$ and $T_2 \rightsquigarrow^* T'$.*

The proof requires the ability to address specific *positions* in a variational type. A position p is given by a path from the root of the type to a particular node, where a path is represented by a sequence of values L and R, indicating whether to enter the left or right branch of a function or choice type. The root type is addressed by the empty path ϵ. We use $T|_p$ to refer to the type at position p in type T. For example, given $T = \texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int}\rangle$, we can refer to the component types of T in the following way.

$$\begin{aligned}
T|_\epsilon &= \texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int}\rangle\\
T|_L &= \texttt{Int}\\
T|_R &= A\langle \texttt{Bool}, \texttt{Int}\rangle\\
T|_{RL} &= \texttt{Bool}\\
T|_{RR} &= \texttt{Int}
\end{aligned}$$

We use $T[T']_p$ to indicate the substitution of type T' at position p in type T. For example, given the same T as above, $T[\texttt{Bool}]_R = \texttt{Int} \to \texttt{Bool}$. We use $\mathscr{P}(T)$ to refer to the set of all positions in T.

We also need a way to abstractly represent the application of a simplification rule. We use $l \rightsquigarrow r$ to represent an arbitrary simplification rule from Figure 6. We represent applying that rule somewhere in type T by giving a position p and a substitution σ indicating how to instantiate it. Before we apply the rule, it must be the case that $T|_p = l\sigma$. The result of applying the rule will be $T[r\sigma]_p$. For example, given $T = \texttt{Int} \to A\langle \texttt{Bool}, \texttt{Bool}\rangle$, we can apply the S-C-IDEMP rule ($l = A\langle x, x\rangle$, $r = x$) at $p = R$ with the substitution $\sigma = \{x \mapsto \texttt{Bool}\}$, resulting in $T' = \texttt{Int} \to \texttt{Bool}$ (note that we assume the dimension name in the simplification rule is instantiated automatically).

PROOF. Given type T, assume that some rewrite rule $l_1 \rightsquigarrow r_1$ can be applied at position p_1 with substitution σ_1, and another rewrite rule $l_2 \rightsquigarrow r_2$ can be applied at position p_2 with substitution σ_2. Then $T_1 = T[r_1\sigma_1]_{p_1}$ and $T_2 = T[r_2\sigma_2]_{p_2}$. Then we must show that there is always a T' such that $T_1 \rightsquigarrow^* T'$ and $T_2 \rightsquigarrow^* T'$. There are three cases to consider.

First, the two simplifications are *parallel*. This occurs when neither p_1 or p_2 is a prefix of the other. It represents simplifications that are in different parts of the type and therefore independent. The proof for this case is shown in the left graph in Figure 16. If we apply $l_1 \rightsquigarrow r_1$ first, we can reach T' by next applying $l_2 \rightsquigarrow r_2$, and vice versa. This situation is encountered, for example, when we must choose between the S-F-ARG and S-F-RES rules. Intuitively, it does not matter whether we first simplify the argument type or result type of a function type. The situation is also encountered when choosing between the S-C-DOM1 and S-C-DOM2 rules, and the S-C-ALT1 and S-C-ALT2 rules.

Second, one simplification may *contain* the other. This occurs when p_1 is a prefix of p_2 and $l_2\sigma_2$ is contained in the range of σ_1 (the case where p_2 is a prefix of p_1 is dual). For example, consider the following type T.

$$T = A\langle B\langle \texttt{Int}, \texttt{Bool}\rangle, C\langle \texttt{Int}, \texttt{Int}\rangle\rangle$$

We can apply the S-C-SWAP1 rule, $A\langle B\langle x, y\rangle, z\rangle \rightsquigarrow A\langle B\langle x, z\rangle, B\langle y, z\rangle\rangle$, at position $p_1 = \epsilon$ with the substitution $\sigma_1 = \{x \mapsto \texttt{Int}, y \mapsto \texttt{Bool}, z \mapsto C\langle \texttt{Int}, \texttt{Int}\rangle\}$. Or we can apply the S-C-IDEMP rule, $C\langle w, w\rangle \rightsquigarrow w$, at position $p_2 = R$ with the substitution $\sigma_2 = \{w \mapsto \texttt{Int}\}$. Note that $l_2\sigma_2 = D\langle \texttt{Int}, \texttt{Int}\rangle$, which is in the range of σ_1. The proof for this example is illustrated in the right graph of Figure 16, the labels (1) and (2) indicate the number of times the associated rule must be applied. In general, if the variable at $l_1|_{p_2}$ occurs m times in l_1 and n times in r_1, then we

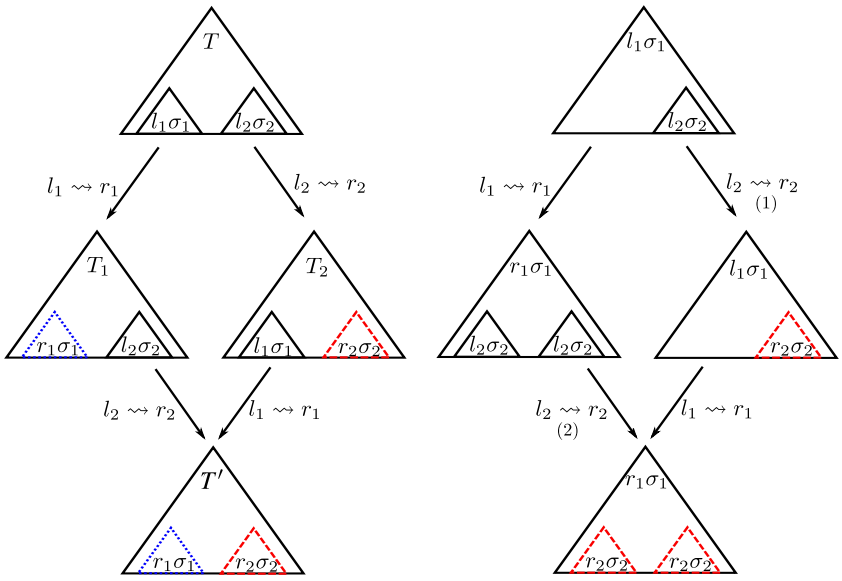

Fig. 16. Proof of local confluence.

need to apply the $l_2 \rightsquigarrow r_2$ rule n times in the left branch of the graph and m times in the right branch. Intuitively, this case arises when the simplifications are conceptually independent, but one is nested within the other. If we apply the outer simplification first, it may increase or decrease the number of times we must apply the inner one (and vice versa). Many combinations of rules can lead to this situation.

Third, the simplifications may *critically overlap*. This occurs when p_1 is a prefix of p_2 and there is some $p \in \mathscr{P}(l_1)$ such that $l_1|_p$ is not a variable and $l_1|_p\sigma_1 = l_2\sigma_2$. For example, consider the following type T.

$$T = C\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, B\langle \texttt{Bool}, \texttt{Int}\rangle\rangle$$

Then both S-C-SWAP1 and S-C-SWAP2 are applicable at $p_1 = p_2 = \epsilon$. To prove that our choice between these rules doesn't matter, we need to compute the critical pairs between the two rules and decide the joinability of all such critical pairs [Baader and Nipkow 1998]. If all critical pairs are joinable, then the two rules are locally confluent, otherwise they are not. The critical pairs are computed as follows. Given any $p \in \mathscr{P}(l_1)$ such that $l_1|_p$ is not a type variable, compute the mgu for $l_1|_p \equiv^? l_2$ as θ, then $r_1\theta$ and $l_1\theta[r_2\theta]_p$ form a critical pair. We show the proof process for the two S-C-SWAP rules below.

First, we rewrite these rules in the following way, so they do not share any type variables. (We also instantiate the dimension names and eliminate the premises.)

S-C-SWAP1	S-C-SWAP2
$C\langle A\langle x,y\rangle,z\rangle \rightsquigarrow A\langle C\langle x,z\rangle, C\langle y,z\rangle\rangle$	$C\langle l, B\langle m,n\rangle\rangle \rightsquigarrow B\langle C\langle l,m\rangle, C\langle l,n\rangle\rangle$

When $p = \epsilon$, the unification problem is $C\langle A\langle x,y\rangle,z\rangle \equiv^? C\langle l, B\langle m,n\rangle\rangle$. The computed mgu θ is given below, where all previously undefined type variables are fresh.

$$\theta = \{x \mapsto C\langle A\langle l,b\rangle,c\rangle, y \mapsto C\langle A\langle d,l\rangle,e\rangle, z \mapsto C\langle f, B\langle m,n\rangle\rangle\}$$

The critical pair consists of the following two types.

1. $A\langle C\langle C\langle A\langle l,b\rangle,c\rangle, C\langle f, B\langle m,n\rangle\rangle\rangle, C\langle C\langle A\langle d,l\rangle,e\rangle, C\langle f, B\langle m,n\rangle\rangle\rangle\rangle$
2. $B\langle C\langle l,m\rangle, C\langle l,n\rangle\rangle$

This pair is joinable by simplifying the first component of the pair into the second, as demonstrated below.

$$\begin{aligned}
&A\langle C\langle C\langle A\langle l,b\rangle,c\rangle,C\langle f,B\langle m,n\rangle\rangle\rangle,\\
&\quad C\langle C\langle A\langle d,l\rangle,e\rangle,C\langle f,B\langle m,n\rangle\rangle\rangle\\
&= A\langle C\langle l,B\langle m,n\rangle\rangle,C\langle l,B\langle m,n\rangle\rangle\rangle && \text{S-C-DOM1 and S-C-DOM2}\\
&= C\langle l,B\langle m,n\rangle\rangle && \text{S-C-IDEMP}\\
&= B\langle C\langle l,m\rangle,C\langle l,n\rangle\rangle && \text{S-C-SWAP2}
\end{aligned}$$

When $p = L$, the unification problem is $A\langle x,y\rangle \stackrel{?}{\equiv} C\langle l,B\langle m,n\rangle\rangle$, and the computed mgu θ is given below.

$$\theta = \{x \mapsto A\langle C\langle l,B\langle m,n\rangle,a\rangle\rangle, y \mapsto A\langle b,C\langle l,B\langle m,n\rangle\rangle\rangle\}$$

The critical pair consists of the following two types.

1. $A\langle C\langle A\langle C\langle l,B\langle m,n\rangle,a\rangle\rangle,z\rangle,C\langle A\langle b,C\langle l,B\langle m,n\rangle\rangle\rangle,z\rangle\rangle$
2. $C\langle B\langle C\langle l,m\rangle,C\langle l,n\rangle\rangle,z\rangle$

This pair is joinable by simplifying both components into the type $C\langle l,z\rangle$, as demonstrated below.

$$\begin{aligned}
&A\langle C\langle A\langle C\langle l,B\langle m,n\rangle,a\rangle\rangle,z\rangle,\\
&\quad C\langle A\langle b,C\langle l,B\langle m,n\rangle\rangle\rangle,z\rangle\\
&= A\langle C\langle l,z\rangle,C\langle l,z\rangle\rangle && \text{S-C-DOM1 and S-C-DOM2}\\
&= C\langle l,z\rangle && \text{S-C-IDEMP}
\end{aligned}$$

$$\begin{aligned}
&C\langle B\langle C\langle l,m\rangle,C\langle l,n\rangle\rangle,z\rangle\\
&= C\langle B\langle l,l\rangle,z\rangle && \text{S-C-DOM1 and S-C-DOM2}\\
&= C\langle l,z\rangle && \text{S-C-IDEMP}
\end{aligned}$$

The proofs for other critically overlapping rules can be constructed similarly. □

A.3.3. Proof of Lemma 4.5

LEMMA 4.5 (TERMINATION). *Given any type T, $T \rightsquigarrow^* T'$ is terminating.*

PROOF. To support this proof, we use a tuple $(a,(u,r),d,i)$ to measure how normalized a type is. When this tuple is $(0,(0,0),0,0)$, the type is fully normalized and no rule in Figure 6 can be applied. Otherwise, the type is not fully normalized and some rule applies. We then divide all the rules in Figure 6 into four groups and show that the first group decreases a, the second group decreases the pair (u,r) without increasing a, the third group decreases d without increasing a or (u,r), and the fourth group decreases i without increasing any other component of the tuple.

The components of $(a,(u,r),d,i)$ are defined as follows.

(1) The component a denotes the number of choice types that are nested directly or indirectly in arrow types. For example, $a = 3$ for the type $A\langle B\langle\tau_1,\tau_2\rangle,\tau_3\rangle \to C\langle\tau_4,\tau_5\rangle$.
(2) The pair (u,r) captures nested choice types that violate the ordering constraint $\preceq$ on dimension names. The component u denotes the *unique* pairs of inverted dimension names, while r is the *total* number of nested choices that violate this constraint. There are some subtleties to computing these values, which are described below.
(3) The component d denotes the number of dead alternatives—alternatives that cannot be selected because of choice domination.
(4) Finally, i denotes number of idempotent choice types—choice types where both alternatives are the same.

To compute the pair (u,r), we cannot just count the number of inverted choice types directly since the process of hoisting can mask intermediate progress if represented in this way. For example, given the type $T = C\langle B\langle T_1,T_2\rangle,A\langle T_3,T_4\rangle\rangle$, an application of S-C-SWAP1 yields $T' = B\langle C\langle T_1,A\langle T_3,T_4\rangle\rangle,C\langle T_2,A\langle T_3,T_4\rangle\rangle\rangle$, in which the choice type in B has been correctly hoisted above the choice type in C. However, T contains just 2 inverted dimension names ($C \not\preceq B$ and $C \not\preceq A$) while T' contains 2 unique inversions ($B \not\preceq A$ and $C \not\preceq A$) and 4 total inversions (since both unique inversions appear twice). Thus, the naive measure fails to capture the progress made transforming T to T', and in fact suggests that we regressed despite having resolved one of the original inversions. Therefore, we instead compute u as (a) the number of unique inversions, plus (b) the binomial coefficient $\binom{n}{2}$ where n is the number of unique inversions involving the root choice type. This additional component captures the combinations

of inversions that could be created during the hoisting process. We compute r in the same way but without limiting ourselves to unique inversions. Using this metric, the transformation of T to T' reduces (u,r) from $(3,3)$ to $(2,4)$.

Now we divide the rules in Figure 6 into four groups. Since the rules S-F-ARG, S-F-RES, S-C-ALT1 and S-C-ALT2 are congruence rules, they have no direct effect on the normalization metric. We divide the remaining rules as follows.

(1) The first group contains the rules S-F-C-ARG and S-F-C-RES. Whenever one of these rules is applied, an arrow type is pushed into a choice types, reducing a.

(2) The second group contains the rules S-C-SWAP1 and S-C-SWAP2. Applying these rules will nor increase a since swapping choices will not generate new choice types and will not push choice types into arrow types. Applying these rules will either decrease u or leave u unchanged and decrease r. For example, applying S-C-SWAP1 to $T = C\langle B\langle\cdots\rangle, A\langle\cdots\rangle\rangle$ leads to the type $T' = B\langle C\langle\cdots, A\langle\cdots\rangle\rangle, C\langle\cdots, A\langle\cdots\rangle\rangle\rangle$. Assume that when computing u for T, parts (a) and (b) are n_1 and n_2, respectively, where $n_2 = \binom{n_3}{2}$. This means that n_1 is the total number of unique inversions in T and n_3 is the number of unique inversions involving dimension C. Similarly, when computing u for T', assume parts (a) and (b) are n'_1 and n'_2, respectively, where $n'_2 = \binom{n'_3}{2}$. Then n'_1 can be computed as follows:

$n'_1 = -1$	previously $C \not\preceq B$, now B is hoisted out
$+ n_{23}$	number of choices nested in A that were inverted with B
$+ n_1$	remaining nestings are unchanged

Since $B \preceq C$, the choices inverted with C in T may be no longer inverted with B in T'. Thus, $n'_3 \le n_3 - 1$ since B is no longer nested in C. Therefore, $n'_2 \le \binom{n_3}{2}$. Moreover, since there are n_3 choices in T inverted with C, there are no more than $n_3 - 1$ choices in A that are inverted with C. Combining with the fact that $B \preceq C$, then there are fewer than $n_3 - 1$ choices that are inverted with C in T'. So we have $n_{23} \le n_3 - 1$.

The change of u from T to T', denoted as χ, can be computed as follows,

$$\begin{aligned}
\chi &= n_1 + n_2 - n'_1 - n'_2 \\
&\ge n_1 + C^2_{n_3} - n'_1 - C^2_{n_3-1} \\
&= (n_1 - n_1 + 1 - n_{23}) + (C^2_{n_3} - C^2_{n_3-1}) \\
&= (1 - n_{23}) + (n_3 - 1) \\
&= n_3 - n_{23} \\
&\ge 1
\end{aligned}$$

Thus, after hoisting, u decreases at least by 1. The proof for the case that $n_2 \le 2$ is simple and is omitted here.

When T is a part of a larger type, then there are two cases. First, if the larger type does not have B nested in C in elsewhere, then it is clear that u will decrease by at least 1 for the larger type. Otherwise, if B is nested in C elsewhere, then u may stay the same. However, we can prove that r decreases at least by 1 similarly to the case for u shown in detail above. Intuitively, r decreases because swapping B out of C has removed the reversed pair between C and B.

(3) The third group includes the rules S-C-DOM1 and S-C-DOM2. Applying the rules in this group will not increase a since they don't create new choices and they don't push down choice types into arrows. Applying them also doesn't increase (u,r) since choice orderings are not swapped. Whenever one of the two rules is applied, at least one dead alternative is removed. Thus, the rules will only decrease d.

(4) The fourth and final group contains the rule S-C-IDEMP. Applying this rule will remove a choice whose alternatives are the same, therefore it may decrease a, (u,r), or d, but it can never increase these values. Whenever this rule is applied, at least one idempotent choice will be eliminated. Therefore, it strictly decreases i.

If we define an ordering relation on $(a,(u,r),d,i)$ based on the ordering relation of each component, where the components are ordered from most-significant (a) to least (i), then we have demonstrated that the metric $(a,(u,r),d,i)$ is strictly decreasing by applying the simplification rules. This process terminates when $(a,(u,r),d,i)$ reaches $(0,(0,0),0,0)$, completing the proof. □