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 very 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 the first problem and addressed only the problem of type checking. In this work we tackle the more general problem of variational type inference. We begin by introducing 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 can have variational types, and a variational type inference algorithm for inferring these 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 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. The inference algorithm is an extension of algorithm \( \mathcal{W} \). We replace the standard unification algorithm with an equational unification algorithm for variational types. We show that the unification problem is decidable and unitary, and that the type inference algorithm computes principal types. 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...
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.

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, a single type error in the variational program is likely to cause type errors in a huge number of program variants. These errors will also be in terms of particular variants rather than the actual variational source code. Similarly, the inferred types of a program variant will not correspond to the variational code, making them not useful as a way to describe and understand a 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, which is absent from other approaches. 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, and supporting understanding of variational software by characterizing the variational structure of the encoded types.

There are several different approaches to managing variation in software, which differ in their ability to support the generalization of traditional static analyses to the variational case. In Section 2 we describe and compare these approaches. Specifically, we will argue that annotative approaches to managing variation offer the best opportunities to develop static analysis techniques that are general and can be applied to a variety of languages.

Following from this, in Section 3 we present the variational lambda calculus (VLC), a conservative extension of the lambda calculus with constructs for introducing and organizing variation with annotations. The extensions are based on our own previous work on representing variation [Erwig and Walkingshaw 2011]. VLC provides simple, direct, and highly-structured support for static variability in lambda calculus expressions, enabling arbitrarily fine-grained and widespread variation. We demonstrate this with a few examples in Section 3, emphasizing how variability can impact the types of functions and expressions. In addition to its use in this paper, VLC can serve more generally as a formal foundation for theoretical research on variational software.

In Section 4 we develop a type system for VLC that relates variational types to VLC expressions. The structure of variational types is given in Section 4.1, and the typing relation is given by a set of typing rules in Section 4.2. While the representation of variation in programs and types is based on the same concepts, variational types have a strictly simpler structure. The reason for this design decision is explained in Section 4.3.

The crucial component of the variational 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 5 together with a corresponding rewriting relationship, normal forms for variational types, and their properties.

In Section 6 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. This is the most important theoretical result of the type system.

Additional features are needed to apply this work to real-world functional programming languages like Haskell, for example, in order to support variational data types. Fortunately, these extensions are fairly straightforward. We outline the necessary steps in Section 7 and demonstrate this by extending the type system to support sum types.

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 8 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 9 we define the type inference algorithm as an extension of algorithm $W$ and demonstrate that the algorithm is sound, complete, and has the principal typing 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 10 we characterize the efficiency gains of variational type inference over the brute-force approach, then demonstrate these gains in a series of simple experiments.

In the context of variational types, the problem of partial type inference becomes very important. Specifically, we would like to be able to type some variants of a variational program, even if not all variants are type correct. We have started to investigate an extension of the type system presented in this paper to support partial typing [Chen et al. 2012]. The results and techniques extend, but are mostly orthogonal to, the results of this paper. We discuss that paper along with other related work in Section 11. We present conclusions and ideas for future work in Section 12.

## 2. 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.

Each of the three approaches to variation management has its own strengths and weaknesses. These are summarized in Figure 1 and will be discussed in the rest of this section. Each of the qualities in the table are expressed positively (that is, “High” is always good), but the qualities are not weighted equally, and their relative importances will vary depending on the user and task. We will focus in this discussion on why annotative approaches, in particular, are worthy of study, and how they support the development of general static analysis techniques for variational software.

First, annotative approaches have the highest degree of language independence. CPP, for example, can be applied to the source code for almost any textual programming language (or any other text file), as long as its syntax does not interfere with the #-notation of CPP. Software projects usually consist of several different artifact types (source code, build scripts, documentation, etc.). Language independence makes it easy to manage and synchronize variation across all of these different artifact types, and trivial to incorporate variation in new artifact types. Some compositional approaches, like the AHEAD tool suite [Batory et al. 2004], provide a degree of language independence through parameterization. By identifying an appropriate representation of a refinement for each object language type, and implementing an operation for composing refinements, the system can be extended to support new object languages. Metaprogramming approaches are usually tightly coupled to their object languages.

Of course, language independence comes at a cost. Since CPP knows nothing about the underlying object language, it is easy to construct variational programs in which not all variants are syntactically correct, and this can usually not be detected until the variant is generated and run through a compiler. This, in turn, has been addressed by annotative variation tools like CIDE [Kästner et al. 2008], and in our own previous work on the choice calculus [2011], by operating on the abstract syntax tree of the object language. This maintains a high degree of language independence while providing structure that ensures the syntactic correctness of all program variants.

Compositional approaches are strongly motivated by traditional software engineering pursuits like separation of concerns (SoC) and stepwise refinement [Batory et al. 2003; Batory et al. 2004; Mezini and Ostermann 2004; Prehofer 1997]. These represent the ideals that the code corresponding to a feature should be in some way modular, understandable independently of other features, and able to be added without changing the existing source code. Neither annotative approaches nor metaprogramming techniques directly support SoC, though Kästner and Apel [2009] propose the idea of a “virtual separation of concerns” (VSoC), where
some of the benefits of SoC are brought to annotative systems with tool support for working with projections of annotated artifacts.

VSoC is only possible because annotative approaches provide an explicit representation of the variation structure of variational software. This is an important (but often overlooked) feature of annotation-based variation management since it supports the direct manipulation and analysis of variation in the artifact. For example, in our work on the choice calculus we have discussed the transformation of variational programs in some detail [2011], including variant-preserving transformations, projections (needed for VSoC), and techniques for eliminating dead code. We have also identified normal forms for variational structures and transformations to achieve them. Such basic analyses like counting or enumerating variants, characterizing relationships between variants, or ensuring that a variational program conforms to a specification are trivial in structured annotative representations, but difficult or impossible in the other approaches.

Related to this is the issue of variation visibility, which is the ability to determine which parts of the artifact are varied and how. This is difficult in metaprogramming approaches because it is not always clear which macros represent variational concerns. It is very difficult in compositional approaches since the addition of modularized refinements (such as aspects) can have far-reaching effects on existing, previously non-variational code.

Finally, annotative approaches are least integrated with the existing static analyses, such as type checking, supported by the object languages they vary. The usual technique is to type check individual variants after they are generated and the annotations of the metalanguage have been removed. The problems with this approach are described in Section 1. In contrast, basic analyses are usually provided for the language extensions required by compositional approaches, such as mixins and aspects. However, checking the successful composition of these components into a consistent program variant is a difficult problem and still an active area of research (see Section 11.2).

Some metaprogramming systems are already fully integrated with the type systems of their underlying object languages. For example, if a MetaML program is type correct, all of the variants it describes are also type correct. However, this nice property comes at a high price. Many useful variations will be rejected by the MetaML type checker since it requires variants to be of compatible types. More lenient metaprogramming systems, such as Template Haskell, are more expressive but do not offer the guarantee that all variants are type correct [Shields et al. 1998]. Typing these more flexible metaprogramming systems therefore suffers from the same problems as the status quo among annotative approaches.

Thus, while annotative approaches are currently least integrated with the existing static analyses of their object languages, the problem is by no means solved in the context of competing approaches. Furthermore, annotative approaches provide a better 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 best 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.

Finally, annotative approaches are also worth studying simply because they are so widely used. While CPP-annotated code does not, in general, provide the same structural guarantees that the choice calculus and CIDE provide, it can provide many of the same benefits if used in a disciplined way [Kenner et al. 2010]. Recent research in this area is discussed in Section 11.2.

1Structured annotative representations include CIDE and the choice calculus, but not CPP [Kästner et al. 2008; Erwig and Walkingshaw 2011].
3. VARIATIONAL LAMBDA CALCULUS

After providing a quick overview of the main constructs of our chosen variation representation in Section 3.1, we will define the syntax and semantics of variational lambda calculus (VLC) in Sections 3.2 and 3.3. A type system for variational programs will then be presented, based on VLC, in Section 4.

3.1. 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 introducing variation points in the code. Dimensions are used to synchronize and scope related choices, and to provide structure to the variation space.

As an example, consider that we have two different ways to represent some part of a program’s state. In one version of the program the state is binary and we use a boolean representation; in another version of the program there are more possible states, so we use integers. We may thus have to choose in an assignment between different values of different types, which is expressed in the choice calculus as a choice.

\[ x = \text{Rep}(\text{True}, 1) \]

The two different alternatives are tagged with a name, \( \text{Rep} \), that stands for the decision to be made about the representation and that allows the synchronization of different choices at different places in the program. 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.

\[ \text{Rep}(x = \text{True}, x = 1) \]

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.

\[ \text{if Rep(not } x, x < 1) \text{ then ...} \]

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 name comes critically into play. It marks and synchronizes all choices in the program that are affected by a single decision; in the case of \( \text{Rep} \), this is the question of whether to choose the boolean or integer representation. Such a decision is represented in the choice calculus as a dimension declaration, which introduces the name of the decision plus names to select the different alternatives, and the scope in the program where this decision is effective. For our example, the dimension declaration would look something like this.

\[ \text{dim Rep(binary, n-ary) in ...} \]

If we select the option binary from the dimension \( \text{Rep} \), the first alternatives of all \( \text{Rep} \) choices bound by this declaration will be selected. Correspondingly, if we select n-ary, all of the second alternatives will be selected.

So what is the type of \( x \)? It can be either \( \text{Bool} \) or \( \text{Int} \), depending on the decision made for the dimension \( \text{Rep} \). We could therefore express the type of \( x \) also using a choice.

\[ x : \text{Rep(Bool, Int)} \]
Dimensions can have more than two options, but all choices bound by a dimension must have the same number of alternatives. Of course, a variational program can have many different dimensions that can be arbitrarily nested. The choice calculus is principally agnostic about the language that is being annotated by choices and dimensions; it works with any kind of tree-structured artifact. The choice calculus also provides a static sharing construct for factoring out code common to multiple variants.

Note that, unlike CPP, in which the ability to vary arbitrary text makes it possible to write code in which some variants are syntactically correct and some are not, the choice calculus guarantees the syntactic correctness of all program variants by working within the object language’s abstract syntax tree, rather than varying plain text.

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 the lambda calculus. Equivalently, VLC can be viewed as a conservative extension of the lambda calculus with the choice, dimension, and static sharing constructs from the choice calculus.

3.2. VLC Syntax

The syntax of VLC is given in Figure 2. For simplicity of the presentation of typing and equivalence rules, we limit dimensions and choices to the binary case. It is easy to simulate each n-ary dimensions by n−1 nested binary ones. The first four VLC constructs, constant, variable, abstractions, and application are as in other lambda calculi. The dimension, choice, binding, and reference constructs are all from the choice calculus. The first two were explained in the previous section, the sharing construct is much like a let construct in lambda calculus, except that it works on the annotation level, that is, it is used to share common subexpressions across different program variants. This sharing is expanded only after a specific program variant has been selected.2

If all choices are bound by corresponding dimension declarations and 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 dimensions, choices, bindings, or references (that is, it is a regular lambda calculus expression, extended with constants), we say that the expression is plain.

3.3. Semantics

Conceptually, a VLC expression represents a set of related lambda calculus expressions. It is important to stress that the choice calculus constructs in VLC describe static variation in lambda calculus. That is, we will not extend the semantics of lambda calculus to incorporate dimensions, choices, and the like. Rather, the semantics of a VLC expression is a mapping

---

2We call this 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.
that describes how to produce each variant (plain) lambda calculus expression from the VLC expression.

In order to select a particular variant, we must keep selecting tags from dimensions until we produce an expression with no dimensions or choices. We write $[e]_{D,t}$ for the selection of tag $t$ from dimension $D$ in expression $e$. Tag selection is defined explicitly in our previous work [Erwig and Walkingshaw 2011] and not repeated here, though we will describe it briefly. First, the topmost-leftmost declaration for dimension $D$ is found, $\dim A(t_1, t_2)$ in $A(e_1, \dim B(t_3, t_4)$ in $B(e_2,e_3))$. If $t = t_1$, we replace every choice bound by that dimension with its first alternative, otherwise if $t = t_2$, we replace every such choice with its second alternative. We call the selection of one or several tags a decision and a decision that eliminates all dimensions and choices from a VLC expression a complete decision.

After a complete decision we are not necessarily left with a plain expression because of the potential presence of share expressions and share-bound variables. We can eliminate these by performing variable substitution in the usual way. In the following we assume that this is done implicitly after any complete decision.

Since the variants represented in a VLC expression are uniquely identified by the decisions that produce them, we define the semantics of a VLC expression $e$ to be a mapping from complete decisions to plain expressions, and we write this as $[[e]]$. Furthermore, to simplify the structure of the semantics and maintain consistency with the previous work, we consider only decisions where tags are listed in the order that their corresponding dimensions are encountered in a pre-order traversal of the expression. To illustrate this, we give the semantics of an expression with a dependent dimension explicitly below. In this example $e_1$, $e_2$, and $e_3$ are plain expressions.

$$[[\dim A(t_1,t_2) \text { in } A(e_1, \dim B(t_3,t_4) \text { in } B(e_2,e_3))]] =$$

$$(\{[A.t_1],e_1\},\{[A.t_2,B.t_3],e_2\},\{[A.t_2,B.t_4], e_3\})$$

Note that tags in dimension $A$ always occur before tags in dimension $B$ in decisions, and that dimension $B$ does not appear at all in the first entry since it is eliminated by the selection of the tag $A.t_1$.

4. TYPE SYSTEM

In this section we present a type system for VLC. Based on a representation of variational types, given in Section 4.1, we will present the typing rules for VLC expressions in Section 4.2. In Section 4.3 we examine and reject an alternative representation of variational types.

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 rules. This technical aspect is provided in Section 5. Finally, in Section 6 we present our main result, which says that typing is preserved over tag selection.

4.1. Variational Types

As the example in Section 3.1 demonstrates, describing the type of variational programs requires a similar notion of variational types. The representation of variational types for VLC
Extending Type Inference to Variational Programs

Rather than requiring that the type of the argument and the argument type of the function

The notation D to choice expressions; for example, the expression

ture variability between types. Choice types often (but do not always) correspond directly
contain only these three constructs. Non-plain types contain
plain types are similar to other type systems. We extend the notion of plainness to types, defining that
is given in Figure 3. The meanings of constant types, type variables, and function types
in the presence of variation, however, so the
T-A
explicit type annotations [Pierce 2002]. The typing of applications is slightly more complex
ceed to the most important part of the type system—the mapping from VLC expressions to

is a constant of type \( \tau \)

Δ, Λ, Γ ⊢ e : τ

\( \Delta, \Lambda, \Gamma \vdash e : T' \)

Δ, Λ, Γ ⊢ \( \lambda x. e : T' \rightarrow T \)

\( \Delta, \Lambda, \Gamma \vdash x : T \)

Δ, Λ, Γ ⊢ e : T''

Δ, Λ, Γ ⊢ e' : T''

\( T'' \equiv T'' \rightarrow T \)

Δ, Λ, Γ ⊢ e : T

Δ, Λ, Γ ⊢ e' : T

Δ, Λ, Γ ⊢ \( \text{share} v = e' \) in e : T

Δ, Λ, Γ ⊢ \( \text{dim} D(t''') \) in e : T

Δ, Λ, Γ ⊢ D(t''') in e : T

Δ, Λ, Γ ⊢ e_1 : T_1

Δ, Λ, Γ ⊢ e_2 : T_2

\( \Delta(D) = D' \)

Δ, Λ, Γ ⊢ D(e_1, e_2) : D'(t', t_2)

\( \Delta(D) = D' \)

Δ, Λ, Γ ⊢ \( \text{dim} D(t''') \) in e : T

Fig. 4. VLC typing rules.

is 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 capture
variability between types. Choice types often (but do not always) correspond directly
to choice expressions; for example, the expression \( D(2, \text{true}) \) has the corresponding choice
type \( D(\text{Int}, \text{Bool}) \). It is important to note, however, that the representation of variation at the
type level is simpler than the representation of variation at the expression level. Specifically,
there is no dimension type construct to bind choice types, so dimension names in choice types
are globally scoped. The use of global dimension names greatly simplifies many aspects of
the type system and will be discussed in depth in Section 4.3. In the next section we pro-
cceed to the most important part of the type system—the mapping from VLC expressions to
variational types.

4.2. Typing Rules

The association of types with expressions is determined by a set of typing rules, given in
Figure 4. These rules also enforce the context-sensitive syntactic constraints on well-formed
VLC expressions, described in Section 3.2, that all choices are bound by corresponding di-
mensions and that all share-bound variable references are bound by corresponding share
expressions.

We use three separate environments in the typing rules, \( \Delta, \Lambda, \Gamma \), each implemented as a
stack. The notation \( E \equiv (k, v) \) 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 \( \Delta \) environment maps a dimension name \( D \) to a fresh (globally previously unused) type-
level dimension name \( D' \). The \( \Lambda \) environment maps a share-bound variable \( v \) to the type of
its bound expression \( T \). Finally, the \( \Gamma \) environment is the standard typing environment for
lambda calculus variables. The use of separate environments for lambda calculus variables
and share-bound variables is required since these variables exist in separate namespaces.

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 environ-
ment \( \Gamma \) 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 slightly more complex
in the presence of variation, however, so the T-APP rule differs from the standard definition.
Rather than requiring that the type of the argument and the argument type of the function

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
are equal, we instead require that they be equivalent using the type equivalence relation ≡. We return to the T-APP rule, show why type equivalence is needed, and define the type equivalence relation in Section 5.

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.

Because dimensions are not directly reflected at the type level, the type of a dimension declaration is simply the type of its scope, and this is captured in the rule T-DIM. Within this scope, the Δ environment is extended to contain a mapping from the dimension’s name D to a fresh dimension name D’ as described above. The second premise of this rule ensures that D’ is a globally unique dimension type name. For simplicity, we can assume that D’ = D wherever possible, and that D’ is derived in a mechanical way from D elsewhere (for example, by drawing the next unique name from the sequence D₁,D₂,…). The T-CHOICE rule ensures that the choice is within the scope of a matching dimension. The type of the choice is then a choice type with the unique dimension name retrieved from the environment, where each alternative in the choice type is the type of the corresponding alternative in the choice expression.

The use of global type-level dimension names has some associated costs. The most obvious and significant is that we lose the one-to-one mapping between expression dimension names and type dimension names. Another minor cost is that we must do some extra administrative work, captured in the T-DIM and T-CHOICE rules, to ensure that type-level dimension names are unique. However, it turns out the first cost is unavoidable even with local dimension type declarations, while the second pales in comparison to the difficulties caused by explicit type-level dimension declarations. We demonstrate each of these claims in the next section.

4.3. Managing Dimensions in Types

The use of global dimension names greatly simplifies the type system for VLC. Because programming language researchers have a reflexive aversion to the term “global”, and since local dimension declarations are an important feature of VLC on the expression level, we devote some space in this section to justifying this important design decision.

For the purposes of the ensuing discussion, consider an extended representation of variational types with the following added construct for locally-scoped, type-level dimensions.

\[ T ::= \ldots \mid \text{dim } D(t,t) \text{ in } T \]

Furthermore, assume that when using this extended type representation, we also use the following modified version of the T-DIM rule that introduces type-level dimension declarations.

\[
\frac{\Delta \equiv (D,D), \Lambda, \Gamma \vdash e : T}{\Delta, \Lambda, \Gamma \vdash \text{dim } D(t₁,t₂) \text{ in } e \text{ : dim } D(t₁,t₂) \text{ in } T}
\]

The variant rule T-DIM* also removes the second premise from T-DIM and adds D directly to the environment. We no longer need to track this value in the environment, strictly speaking, but we add it so that we can use the standard T-CHOICE rule unchanged.

One problem we can observe with the extended type system is the problem of dimension duplication. Consider the following simple example, where \(c₁ : τ₁\) and \(c₂ : τ₂\).

\[(\lambda x.λ f. f \ x \ x) (\text{dim } A(t,u) \text{ in } A(c₁,c₂))\]

This example represents two variants. If we select \(A.t\), we get the expression \((\lambda x.λ f. f \ x \ x) c₁\). If we select \(A.u\), we get \((\lambda x.λ f. f \ x \ x) c₂\). However, the extended type system identifies the following variational type with four type variants!

\[(\text{dim } A(t,u) \text{ in } A(τ₁,τ₂) → \text{dim } A(t,u) \text{ in } A(τ₁,τ₂) → a) → a\]
The problem is that after the dimension type is added to the typing environment $\Gamma$ 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.

$$\text{dim } A \langle t, u \rangle \text{ in } (A \langle t_1, t_2 \rangle \rightarrow A \langle t_1, t_2 \rangle \rightarrow a) \rightarrow a$$

Unfortunately, this solution does not work in general. This is demonstrated by the next example, which cannot be correctly typed using type-level dimension declarations without resorting to dimension renaming.

$$(Ax. \text{dim } A \langle r, s \rangle \text{ in } A \langle c_1, c_2 \rangle) \text{ dim } A \langle t, u \rangle \text{ in } A \langle c_1, c_2 \rangle$$

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

$$\text{dim } A \langle r, s \rangle \text{ in } (A \text{ dim } A \langle t, u \rangle \text{ in } A \langle t_1, t_2 \rangle \rightarrow A \langle t_1, t_2 \rangle \rightarrow a) \rightarrow a$$

Here we have again duplicated the dimension type from the argument. This time, however, we cannot factor out the repeated dimension type because doing so would capture the $A$ choice enclosing the first repeated dimension. We can only represent an equivalent type using locally-scoped dimension types by renaming one of the dimensions.

$$\text{dim } A \langle r, s \rangle \text{ in dim } A' \langle t, u \rangle \text{ in } (A \text{ A' } \langle t_1, t_2 \rangle \rightarrow A' \langle t_1, t_2 \rangle \rightarrow a) \rightarrow a$$

Of course, this breaks the one-to-one mapping from expression-level dimension names to type-level dimension names, which we might have expected locally scoped dimension type could provide. In other words, the scope of dimension declarations at the expression level fundamentally cannot be preserved at the type level, greatly weakening the argument in favor of local dimension types.

In addition, 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 8). Furthermore, the simple factoring transformation shown above often fails in the presence of dependent dimensions and dimensions of the same name, leaving the correct typing of many expressions to resort to dimension renaming. Global dimension names offer an easy solution to the first problem—dimension duplication is trivially a non-issue when all type-level dimensions are unique and globally defined. That global dimensions also sometimes require deviating from the provided expression-level dimension names is moot since the same is also true of a type representation with explicit dimension declarations.

Although we assume here that dimension types are global, we can imagine a simple module system for explicitly managing this namespace. Dimension names would be “global” within modules, and any collisions between modules would have to be manually resolved when combining them. This is similar to the way type names are managed in Haskell [Peyton Jones 2003].

5. TYPE EQUIVALENCE

In this section we return to the discussion of the T-APP rule from Section 4.2. 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.

$$\text{dim } A \langle a_1, a_2 \rangle \text{ in succ } A \langle 1, 2 \rangle$$
By the T-DIM typing rule, the type of this expression will be the type of the application in the scope of the dimension. The LHS of the application, succ, has type \( \text{Int} \rightarrow \text{Int} \); the RHS, \( A(1,2) \), has type \( A(\text{Int}, \text{Int}) \). Since \( \text{Int} \neq A(\text{Int}, \text{Int}) \), 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 (\( \text{succ} \ 1 \) and \( \text{succ} \ 2 \) are perfectly well-typed (both have type \( \text{Int} \)).

Although the types \( \text{Int} \) and \( A(\text{Int}, \text{Int}) \) 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 \( D(T_1, T_2) \) 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 in the same dimension. For example, \( D(D(1,2), 3) \) is semantically equivalent to \( D(1, 3) \) 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 in the same dimension 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 [Erwig and Walkingshaw 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.

Fig. 5. VLC type equivalence.
The type does not contain dominated choices. For example, the type $\text{Int} \rightarrow \text{Bool}$ is in normal form, while $A(\text{Int}, a) \rightarrow A(a, \text{Bool})$ is not.

2. The type does not contain dominated choices. For example, the type $A(\text{Int}, \text{Bool}, a)$ is not in normal form. It can be simplified to $A(\text{Int}, a)$, which is in normal form.

3. The nesting of choices adheres to a fixed ordering relation $\prec$ on dimension names. That is, if $T$ is in normal form and $A \prec B$, then no choice type $A(\ldots)$ appears within an alternative of a choice type $B(\ldots)$ in $T$.

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.
We define a relation $\rightsquigarrow$ to rewrite a type into a simpler one and use the reflexive, transitive closure of this relation $\rightsquigarrow^*$ to transform types into normal forms. The type simplification rewriting rules are shown in Figure 6. The S-F-C-Arg and S-F-C-Res rewriting rules are simply directed versions of the corresponding F-C equivalence rules from Figure 5. The S-C-Imp rewriting rule is similarly derived from C-Imp, except that a choice can only be eliminated if the two alternatives are structurally equal. This means that the alternatives of a choice type must be simplified first using the S-C-Imp rewriting rules, which are focused adaptations of the choice equivalence rule; each one simplifies a single alternative.

The Fun equivalence rule is similarly split into two rewriting rules, S-F-Arg and S-F-Res, which simplify the argument type and the result type of a function type, respectively. The S-C-Swap and S-C-Dom rules follow less directly from their corresponding equivalence rules. The S-C-Swap rewriting rules add premises to ensure that dimensions are swapped only when they satisfy the dimension name ordering relation. The S-C-Dom rule uses selection to eliminate arbitrarily nested dominated dimensions.

We can transform any variational type into normal form by repeatedly applying the rewrite rules in Figure 6. Figure 7 shows an example transformation of the type $B(A(r_1, r_2), A(r_1, r_2))$ into its corresponding normal form $A(r_1, r_2)$ (we assume lexicographic ordering for $\preceq$). Note that in the application of the S-C-Swap1 rule, we arbitrarily chose to swap the nested choice in the first alternative. We could have also applied S-C-Swap2, or applied S-C-Imp 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 5.3 below.

**Theorem 5.3 (Confluence).** If $T \rightsquigarrow^* T_1$ and $T \rightsquigarrow^* T_2$, then there exists a $T'$ such that $T_1 \rightsquigarrow^* T'$ and $T_2 \rightsquigarrow^* T'$. 

---

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
A rewriting relation is confluent if it is both locally confluent and terminating. These properties are expressed for the \( \sim^* \) relation below, in Lemmas 5.4 and 5.5. From these two lemmas, Theorem 5.3 follows directly.

**Lemma 5.4 (Local Confluence).** For any type \( T \), if \( T \sim T_1 \) and \( T \sim T_2 \), then there exists some type \( T' \) such that \( T_1 \sim^* T' \) and \( T_2 \sim^* T' \).

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

**Lemma 5.5 (Termination).** Given any type \( T \), \( T \sim^* T' \) is terminating.

**Proof Sketch.** The \( \sim^* \) relation will terminate when we reach a normal form (as defined by the criteria listed earlier in this section) because an expression satisfying these criteria will not match any rule in the \( \sim \) 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 \( \preccurlyeq \) relation on dimension 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. \( \square \)

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 \( \text{norm}(T) \) for the unique normal form of \( T \). We capture the fact that a normal form is a representative of 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 5.6.** \( T \equiv T' \Leftrightarrow \text{norm}(T) = \text{norm}(T') \).

**Proof.** This follows from Theorem 5.3, the fact that \( \sim^* \) is normalizing, and the observation that the \( \equiv \) relation is the reflexive, transitive, symmetric closure of \( \sim \). \( \square \)

This is the essential result needed for checking type equivalence.

### 6. 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 has a plain type that is obtained by essentially the same selection. This result can be proved with the help of the
following lemma, which states that variational typing is preserved over the selection of a single tag.

**Lemma 6.1.** \( \Delta, \Lambda, \Gamma \vdash e : T \implies (e \text{ is plain or } \exists D, t \text{ such that } \Delta, \Lambda, \Gamma \vdash [e]_{D.t} : [T]_{q_e([D,t])}) \).

The function \( q_e \) used in the lemma is a function derived from \( e \) that maps tag sequences to corresponding selector sequences. This is needed since the semantics of variational types use selectors in decisions rather than qualified tags, and since expression-level dimension names may differ from type-level dimension names, as described in Section 4. As an example, consider the expression \( e \) below.

\[
   e = \dim A(t_1, t_2) \in A(\dim A(u_1, u_2) \in A(e_1, e_2), e_3)
\]

Given this expression, here is a possible implementation of the function \( q_e \).

\[
\]

Note that the construction of \( q_e \) must be consistent with how we choose fresh type-level dimension names in the T-DIM typing rule. In the example above we map the expression-level dimension \( D \) to the type-level dimension \( D \) if \( D \) is already fresh, otherwise we add primes to the name until we have a fresh name. Thus, the outer \( A \) dimension in the expression corresponds to the type-level dimension \( A \), while the inner \( A \) dimension corresponds to type-level dimension \( A' \).

In the following proof, we do not use \( q_e \) directly, but assume that it is used implicitly to map each qualified tag \( q \) (of the form \( D.t \)) onto its corresponding selector \( s \).

**Proof sketch of Lemma 6.1.** 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 \( \Delta, \Lambda, \Gamma \vdash e : T \) 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 \( [e e']_q : [T]_s \).

We do this through the following sequence of observations.

1. \( e : T'' \), \( e' : T' \), and \( T'' \equiv T' \rightarrow T \) by the definition of T-App
2. \( [e e']_q = [e]_q [e']_q \) by the definition of \( [.]_q \)
3. \( [e]_q : [T'']_s \) and \( [e']_q : [T']_s \) by the induction hypothesis
4. \( [T'']_s \equiv [T']_s \rightarrow [T]_s \) by 1 and Lemma 5.2
5. \( [T'']_s \equiv [T']_s \rightarrow [T]_s \) by 4 and the definition of \( [.]_s \)
6. Therefore, \( [e e']_q : [T]_s \) by 2, 3, 5, and the definition of T-App

For the T-Choice rule, assume that \( D(e_1, e_2) : D' \langle T_1, T_2 \rangle \). Then we must show that \( [D(e_1, e_2)]_q : [D'(T_1, T_2)]_s \). There are two cases to consider: either \( q \) represents a selection in dimension \( D \), or it does not.

If \( q \) represents a selection in dimension \( D \), then \( s \) will also select from \( D' \). From here, the first case follows directly from the induction hypothesis and the definitions of selection on expressions and types. For example, if \( q \) selects the first tag in \( D \), then selecting the first alternative on both sides of the typing relation leaves us with \( [e_1]_q : [T_1]_s \), which is the induction hypothesis.

If \( q \) does not represent a selection in \( D \), then \( s \) will likewise not select from \( D' \). Applying selection to each side of the typing relation, yields \( D([e_1]_q, [e_2]_q) : D'( [T_1]_s, [T_2]_s) \). Since \( [e_1]_q : [T_1]_s \) and \( [e_2]_q : [T_2]_s \) by the induction hypothesis, the claim follows through a direct application of the T-Choice rule. \( \square \)

By induction it follows that for any sequence of tag selections \( \overline{q} \) that yields a plain expression from \( e \), the corresponding selector sequence \( \overline{s} = q_e(\overline{q}) \) selects the plain expression's
corresponding plain type. Therefore, the following theorem, which captures the type preservation property described at the beginning of this section, follows directly from Lemma 6.1.

**Theorem 6.2 (Type Preservation).** If \( \varnothing, \varnothing, \Gamma \vdash e : T \) and \( (\varnothing, e') \in \{e\} \), then \( \Gamma \vdash e' : T' \) where \( q_\varnothing(\varnothing, e) = \varnothing \) and \( (\varnothing, T') \in \{T\} \).

With Theorem 6.2, the type for any particular expression variant of \( e \) can be easily selected from its inferred variational type \( T \). For example, suppose \( \varnothing, \varnothing, \Gamma \vdash e : T \) with \( T = A\langle T_1, T_2, T_3 \rangle \), then the type of \( e^2 = [[e]]_{A,T_1}A_{T_2} \) is \( [[T]]_{A,T} = T_2 \).

7. EXTENSIONS OF THE BASIC TYPE SYSTEM

To extend the VLC type system from lambda calculus to a fully-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 variation 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 \( \text{inl} \, e \) and \( \text{inr} \, e \) as well as a case expression for pattern matching expressions built using \( \text{inl} \) and \( \text{inr} \) [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 \( \text{inl} \) and \( \text{inr} \) are exactly the same as presented by Pierce [2002], except for the additional environments used in the typing judgment. The typing rule for **case**, which is slightly different, is shown below.

**T-CASE**

\[
\Delta, \Lambda, \Gamma \vdash e_0 : T_1 + T_2 \\
\Delta, \Lambda, \Gamma \vdash (x_1, T_1) \vdash e_1 : T_1' \\
\Delta, \Lambda, \Gamma \vdash (x_2, T_2) \vdash e_2 : T_2' \\
T_1' \equiv T_2' \\
\Delta, \Lambda, \Gamma \vdash \text{case } e_0 \text{ of } \text{inl} \, x_1 \Rightarrow e_1 \mid \text{inr} \, x_2 \Rightarrow e_2 : T_1'
\]

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. 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 rule **S-C**.

\[
\text{Sum} \\
T_1' \equiv T_1' \quad T_2 \equiv T_2' \\
T_1 + T_2 \equiv T_1' + T_2' \\
D\langle T_1, T_2 \rangle \equiv D\langle T_1', T_2 \rangle + D\langle T_1, T_2' \rangle
\]

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

\[
\text{S-S-L} \\
T_1 \sim T_1' \\
T_1 + T_r \sim T_1' + T_r
\]

\[
\text{S-S-R} \\
T_r \sim T_r' \\
T_l + T_r \sim T_l' + T_r'
\]

\[
\text{S-S-C} \\
D\langle T_1 + T_2 + T_2' \rangle \sim D\langle T_1, T_2 \rangle + D\langle T_1', T_2 \rangle
\]

Extensions by 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.

8. UNIFYING VARIATIONAL TYPES

The type inference algorithm for VLC is an extension of the traditional algorithm \( \mu \) 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 8.1, while the unification algorithm that solves it is presented in Section 8.2. In Section 8.3 we formally evaluate the correctness of the unification algorithm, and we analyze its time complexity in Section 8.4. Once this groundwork has been laid, the variational type inference algorithm itself is straightforward. It is is given in Section 9.

8.1. 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.

\[ \sigma(\text{Int}, a) \equiv^? \sigma(b, c) \]  

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.

\[
\begin{align*}
\sigma_1 &= \{a \rightarrow \text{Int}, b \rightarrow \text{Int}, c \rightarrow \text{Int}\} \\
\sigma_2 &= \{b \rightarrow A(\text{Int}, a), c \rightarrow A(\text{Int}, c)\} \\
\sigma_3 &= \{a \rightarrow B(\text{Int}, f), b \rightarrow \text{Int}, c \rightarrow A(\text{Int}, f)\} \\
\sigma_4 &= \{a \rightarrow B(f, \text{Int}), b \rightarrow A(\text{Int}, f), c \rightarrow \text{Int}\} \\
\sigma_5 &= \{a \rightarrow B(d, f), b \rightarrow A(\text{Int}, d), c \rightarrow A(\text{Int}, f)\} \\
\sigma_6 &= \{a \rightarrow B\langle i, d \rangle, A\langle j, f \rangle), b \rightarrow B\langle A(\text{Int}, d), g \rangle, c \rightarrow B\langle h, A(\text{Int}, f)\rangle\}
\end{align*}
\]

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 \(\sigma_6\) 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 \(\sigma_6\), we can produce any other unifier. For example, composing \(\sigma_6\) with \(i \rightarrow d, j \rightarrow f, g \rightarrow A(\text{Int}, d), h \rightarrow A(\text{Int}, f)\) yields \(\sigma_5\), which is in turn the most general among the first five unifiers.

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 makes the problem unitary. Specifically, it 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 8.1.** Given a CT-unification problem \(U\), there is a unifier \(\sigma\) such that for any unifier \(\sigma'\), there exists a mapping \(\theta\) 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 \( \text{Int} \equiv ^1 b, \text{Int} \equiv ^2 c, a \equiv ^1 b, \text{and } a \equiv ^2 c \). 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 8.2.
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 8.2.

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.

### 8.2. Qualified Type Unification Algorithm

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.

\[
A(\text{Int}, a) \equiv ^2 A(a, \text{Bool})
\]

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 \( \{\text{Int} \equiv ^2 a, a \equiv ^6 \text{Bool}\} \), which is unsatisfiable. However, notice that \( \{a \rightarrow A(\text{Int}, \text{Bool})\} \) 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 \( a, A(a, T) \), and \( A(T, a) \), 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 \( \tilde{a}_q \), where \( q \) is the qualification and is given by a set of selectors (see Section 5), rendered as a lexicographically sorted sequence. For example, the type variable \( a \) in \( B(T_1, A(a, T_2)) \) corresponds to the qualified type variable \( \tilde{a}_{AB} \). Likewise, the (non-qualified) unification problem in example (1) can be transformed into the qualified unification problem \( A(\text{Int}, \tilde{a}_q) \equiv ^2 B(b_B, c_B) \), and the problem in example (2) can be transformed into \( A(\text{Int}, \tilde{a}_q) \equiv ^2 A(a_A, \text{Bool}) \).

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.

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

\[
\begin{align*}
    A\langle\text{Int},a_{\bar{A}}\rangle & \equiv \top B\langle b_B,c_B\rangle \\
    A\langle\text{Int},B\langle a_{\bar{A}B},a_{\bar{AB}}\rangle\rangle & \equiv \top B\langle b_B,c_B\rangle \\
    B\langle A\langle\text{Int},a_{\bar{A}B}\rangle,A\langle\text{Int},a_{\bar{AB}}\rangle\rangle & \equiv \top B\langle b_B,c_B\rangle
\end{align*}
\]

Fig. 8. Example of qualified unification.

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.

\[
B\langle A\langle\text{Int},a_{\bar{A}B}\rangle,A\langle\text{Bool},a_{\bar{AB}}\rangle\rangle \equiv \top B\langle a_B,\text{Bool}\rangle
\]

Notice that we must add a qualification to all of duplicated type variables that were originally in the alternative opposite the hoisted choice type but are now nested beneath it, such as the \(a_{\bar{A}}\) 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 \(a \equiv \top A\langle a_A,\text{Int}\rangle\), we first split \(a\) into the choice type \(A\langle a_A,\bar{a}_A\rangle\), then decompose by alternatives. To decompose the problem \(A\langle\text{Int},a_{\bar{A}}\rangle \equiv \top B\langle\text{Int},b_B\rangle\), we can split either \(a_{\bar{A}}\) into a choice in \(B\) or \(b_B\) 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 \(\{b_B \rightarrow A\langle\text{Int},a_{\bar{A}B}\rangle\}\) for this subtask directly. Likewise for \(c_B\) in the second subtask.

To solve a unification problem \(U\), we solve the corresponding qualified unification problem \(Q\), then transform the solution of \(Q\), \(a_Q\), into a solution for \(U\), \(\sigma_U\). Each mapping \(a \rightarrow T\) in \(\sigma_U\) is derived through a process called completion from the related subset of mappings in \(\sigma_Q\), \(\{a_{q_1} \rightarrow T_1, \ldots, a_{q_n} \rightarrow T_n\}\). Each qualified mapping \(a_{q_i} \rightarrow T_i\) essentially 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 \(\{a_A \rightarrow \text{Int}, a_{\bar{A}B} \rightarrow b, a_{\bar{AB}} \rightarrow \text{Bool}\}\), completion might produce the unqualified mapping \(a \rightarrow A\langle\text{Int},B\langle b,\text{Bool}\rangle\rangle\).

Formally, we define completion by folding the helper function \(\text{comp}\) in Figure 9 across the mappings in \(\sigma_Q\). This function produces a partially completed type given (1) a type variable qualification \(q\), (2) the type to store at the choice-type path described by \(q\), and (3) the type that is being completed. The definition of \(\text{comp}\) relies on top-down pattern matching on the first and third arguments (\(\epsilon\) matches the empty qualification), and on a second helper function \(\text{fresh}\) that renames every type variable in its argument type to a new, fresh type variable.
\[ \text{comp}(Dq,T,D(T_1,T_2)) = D(\text{comp}(q,T,T_1),T_2) \]
\[ \text{comp}(\bar{D}q,T,D(T_1,T_2)) = D(T_1,\text{comp}(q,T,T_2)) \]
\[ \text{comp}(Dq,T,T') = D(\text{comp}(q,T,T'),\text{fresh}(T')) \]
\[ \text{comp}(\bar{D}q,T,T') = D(\text{fresh}(T'),\text{comp}(q,T,T')) \]
\[ \text{comp}(\epsilon,T,a) = T \]

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

In the first two cases of \text{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 \text{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 heavily 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 \text{comp}; these can be eliminated by a subsequent normalization step.

Finally, using the above definition of \text{comp}, the completion of \( a \rightarrow T \) from \( \{a_{q_1} \rightarrow T_1, \ldots, a_{q_n} \rightarrow T_n\} \) is defined as follows.

\[ a \rightarrow \text{comp}(q_1,T_1,\text{comp}(q_2,T_2,\ldots \text{comp}(q_n,T_n,b)\ldots)) \]

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 \text{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 \text{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, \text{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 \text{unify} \((T,T')\) implies the definition of both \text{unify} \((T',T)\), as written, and a dual case \text{unify} \((T',T) = \text{unify}(T,T')\).

The definition of \text{unify} relies on several helper functions. The function \text{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, \text{hoist}(A(B(a,Int),Bool),B) yields \( B(A(a,Bool),A(\text{Int},\text{Bool})) \). The function \text{choices} takes a type and returns the set of dimension names of all choice types it contains. The function \text{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, \text{splittable}(A(\text{Int} \rightarrow b,c)) = \{c\}. The function \text{occurs} will be explained below.

When unifying two plain types, we defer to Robinson’s unification algorithm [Robinson 1965]. To unify a type variable with a choice type, we split the type variable as described earlier in this section. To unify two choice types in the same dimension, we decompose the problem and unify their corresponding alternatives. To unify two choice types in different
unify : \( (T_L, T_R) \to \sigma \)
unify\((p, p') = \text{robinson}(p, p') \)
unify\(*\( (a_q, D\( (T_1, T_2) \)) = \text{unify}(D\( (\alpha_{D_q}, a_{D_q}), D\( (T_1, T_2) \)) \)
unify\(*\( (D\( (T_1, T_2), D\( (T'_1, T'_2) \)) = \sigma_1 \leftarrow \text{unify}(T_1, T'_1) \)
\quad \sigma_2 \leftarrow \text{unify}(T_2\sigma_1, T'_2\sigma_1) \)
\quad \text{return } \sigma_1 \circ \sigma_2 \)
unify\(*\( (D_1(T_1, T_2), D_2(T'_1, T'_2)) \mid D_2 \in \text{choices}(T_L) = \text{unify}(\text{hoist}(T_L, D_2), T_R) \)
unify\(*\( (D_1(T_1, T_2), D_2(T'_1, T'_2)) \mid \text{splittable}(T_L) \neq \emptyset \land \)
\quad D_2 \notin \text{choices}(T_L) = a_q \leftarrow \text{splittable}(T_L) \)
\quad \theta \leftarrow \{ a_q \mapsto D_2(\alpha_{D_q}, a_{D_q}) \} \)
\quad \text{return unify}(T_L\theta, T_R) \)
unify\(*\( (D_1(T_1, T_2), D_2(T'_1, T'_2)) \mid \text{splittable}(T_L) = \emptyset \land \text{splittable}(T_R) = \emptyset \land D_1 \notin \text{choices}(T_R) = \text{unify}(D_2(T_L, T_L), T_R) \)
unify\(*\( (g, D\( (T_1, T_2) \)) = \sigma_1 \leftarrow \text{unify}(g, T_1) \)
\quad \text{return } \sigma_1 \circ \text{unify}(g, T_2\sigma_1) \)
unify\(*\( (T \rightarrow T', D\( (T_1, T_2) \)) = \text{unify}(D\( (T \rightarrow T', T \rightarrow T'), D\( (T_1, T_2) \)) \)
unify\(*\( (T_1 \rightarrow T_2, T'_1 \rightarrow T'_2) = \sigma \leftarrow \text{unify}(T_1, T'_1) \)
\quad \text{return } \sigma \circ \text{unify}(T_2\sigma, T'_2\sigma) \)
unify\(*\( (a_q, T \rightarrow T') \mid \text{occurs}(a_q, T_R) = \text{fail} \)
\quad \text{otherwise } = \{ a_q \mapsto T_R \} \)

Fig. 10. The qualified unification algorithm.

dimensions, we try to hoist a choice type so that both types are rooted by a choice in the same dimension. If this is impossible, then a splittable type variable is split into a choice in that dimension, which can then be hoisted. 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. 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. Finally, to unify a type variable with a function type, a process similar to an \text{occurs check} is needed. The operation \text{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.

8.3. 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.

\textbf{Lemma 8.2 (Decomposition).} Let \( T_L = D\( (T'_1, T_2) \) \) and \( T_R = D\( (T'_1, T'_2) \) \). Then \( T_L \equiv \gamma T_R \) is unifiable iff \( T_1 \equiv \gamma T'_1 \) and \( T_2 \equiv \gamma T'_2 \) are unifiable. Moreover, if the problem is unifiable, then \( \sigma_1 \circ \sigma_2 \) is a unifier for \( T_L \equiv \gamma T_R \), where \( \sigma_1 \) and \( \sigma_2 \) are unifiers for \( T_1 \equiv \gamma T'_1 \) and \( T_2 \equiv \gamma T'_2 \), 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 $D(T_1, T_2)$, it is always the case that $\text{vars}(T_1) \cap \text{vars}(T_2) = \emptyset$, by the definition of qualification. Specifically, the qualifier of every type variable in $T_1$ will contain the selector $D$, while the qualifier of every type variable in $T_2$ will contain the selector $\overline{D}$. 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 both variable and choice independent. Variable independence means that when more than one variable is splittable, it does not matter which we choose. Similarly, choice independence means that when we can split a variable into several different choice types, the particular dimension we choose does not matter. In either case we will achieve equivalent unifiers regardless of how we split the type variable.

**Lemma 8.3 (Variable Independence).** Let $T_L = D_1(T_1, T_2)$ and $T_R = D_2(T_1', T_2')$. Assume $a_q, b_r \in \text{splittable}(T_L)$, where qualifiers $q$ and $r$ do not contain selectors in dimension $D_2$. Let $\theta_1 = \{a_q \rightarrow D_2(a_{D_2q}, a_{D_2q})\}$ and $\theta_b = \{b_r \rightarrow D_2(b_{D_2r}, b_{D_2r})\}$. Then unify $(T_L\theta_a, T_R) = \text{unify}(T_L\theta_b, T_R)$.

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 unification 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 type variables. For each choice type $D_i(\ldots)$ 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 $D_i(D_2(T_1, T_2), T_3)$. After applying hoist, we have $D_2(D_i(T_1, T_2), D_i(T_2, T_3))$. 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 $D_2$. This means the number of choice types, 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 8.3 is illustrated in Figure 11 with a small example. In this example, $T_L = D_1(a_q, D_3(T_1, b_r))$ and $T_R = D_2(T_2, T_3)$, where $q$ does not contain qualifiers in $D_2$. If $D_3$ and $D_2$ do not contain a qualifier in $D_2$. In the case of $T_3$ is copied and qualified when a choice type is hoisted over them.

Just as it does not matter which splittable type variable we choose, it also 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 8.4 (Choice Independence).** Let $T_L = D_1(T_1, T_2)$ and $T_R = D_2(T_1', T_2')$. Assume $D_m, D_n \in \text{choices}(T_R)$ and $a_q \in \text{splittable}(T_L)$, where qualifier $q$ does not contain selectors in $D_m$ or $D_n$. Let $\theta_1 = \{a_q \rightarrow D_m(a_{D_mq}, a_{D_mq})\}$ and $\theta_2 = \{a_q \rightarrow D_n(a_{D_nq}, a_{D_nq})\}$. Then unify $(T_L\theta_1, T_R) = \text{unify}(T_L\theta_2, T_R)$.

Proof. The proof strategy for this lemma is similar to that for Lemma 8.3. Again, we only show the proof for one case. Assume $T_L = D_1(a_q, T_1)$ and $T_R = D_2(T_2, D_3(T_3, T_4))$. Figure 12 shows the unification process under two different splitting approaches. We notice that split $a_q$ into choice $D_2$ will finally result in three unification subproblems whereas split it into.
The process described in the proof of Lemma 8.4 is illustrated in Figure 12. In this example, \( T_L = D_1(a_q,T_1) \) and \( T_R = D_2(T_2,T_3,T_3,T_3) \), 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 \( D_3 \), 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 represents 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 first case is equivalent to the two subproblems in the second case that have \( T_2 \) on their RHS. This is because \( T_2 \equiv D_3(T_2,T_2) \).

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 also sound, which means that it always produces mappings that unify its arguments. It is also complete and most general, which means that if there is a unifier \( \sigma \) that unifies two types, \( \sigma \) can be obtained by applying some other mapping to the mgu returned by the unification algorithm. We express each of these results in the following theorems.

**Theorem 8.5 (Soundness).** If \( \text{unify}(T_1,T_2) = \sigma \), then \( T_1 \sigma \equiv T_2 \sigma \).

**Theorem 8.6 (Complete and Most General).** If \( T_1 \sigma = T_2 \sigma \), then \( \text{unify}(T_1,T_2) = \sigma' \) where \( \sigma = \sigma'' \circ \sigma' \) for some \( \sigma'' \).

A proof of Theorem 8.5 is given in the appendix in Section A.2.1.
Extending Type Inference to Variational Programs

A.25

\[
D_1 \langle a_q, T_1 \rangle \equiv^? D_2 \langle T_2, D_3 \langle T_3, T_4 \rangle \rangle \\
\]

\[
D_1 \langle D_2 \langle a_{D_{2q}}, a_{\tilde{D}_{2q}} \rangle, T_1 \rangle \equiv^? D_2 \langle T_2, D_3 \langle T_3, T_4 \rangle \rangle \\
\]

\[
D_2 \langle D_1 \langle a_{D_{2q}}, T_1 \rangle, D_1 \langle a_{D_{2q}}, T_1 \rangle \rangle \equiv^? D_2 \langle T_2, D_3 \langle T_3, T_4 \rangle \rangle \\
\]

\[
D_1 \langle a_{D_{2q}}, T_1 \rangle \equiv^? T_2 \\
D_1 \langle a_{\tilde{D}_{2q}}, T_1 \rangle \equiv^? D_3 \langle T_3, T_4 \rangle \\
\]

\[
D_1 \langle a_{D_{2q}}, T_1 \rangle \equiv^? D_2 \langle T_2, D_3 \langle T_3, T_4 \rangle \rangle \\
\]

\[
D_3 \langle D_1 \langle a_{D_{3q}}, a_{\tilde{D}_{3q}} \rangle, T_1 \rangle \rangle \equiv^? D_2 \langle T_2, D_3 \langle T_3, T_4 \rangle \rangle \\
\]

\[
D_3 \langle D_1 \langle a_{D_{3q}}, T_1 \rangle, D_1 \langle a_{\tilde{D}_{3q}}, T_1 \rangle \rangle \equiv^? D_3 \langle D_2 \langle T_2, T_3 \rangle, D_2 \langle T_2, T_4 \rangle \rangle \\
\]

\[
D_1 \langle a_{D_{3q}}, T_1 \rangle \equiv^? D_2 \langle T_2, T_3 \rangle \\
D_1 \langle a_{\tilde{D}_{3q}}, T_1 \rangle \equiv^? D_2 \langle T_2, T_4 \rangle \\
\]

\[
D_1 \langle a_{D_{2q}}, T_1 \rangle \equiv^? T_2 \\
D_1 \langle a_{\tilde{D}_{2q}}, T_1 \rangle \equiv^? T_2 \\
\]

\[
D_1 \langle a_{D_{2q}}, T_1 \rangle \equiv^? T_3 \\
D_1 \langle a_{\tilde{D}_{2q}}, T_1 \rangle \equiv^? T_4 \\
\]

Fig. 12. Demonstration of choice independence.

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 \textit{comp} function, the qualifiers, the unification problem, and the qualified unification problem.

The following lemma states the expectations for the \textit{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 \textit{comp} is correct.

**Lemma 8.7.** Given a mapping \( \{a_q \rightarrow T'\} \), if \( T = \text{comp}(q, T', b) \) is the completed type (where \( b \) is fresh), then \( \{T\}_q = \{T'\}_q \). More generally, given \( \{a_{q_1} \rightarrow T_1, \ldots, a_{q_n} \rightarrow T_n\} \), if \( T = \text{comp}(q_1, T_1, \text{comp}(q_2, T_2, \ldots, \text{comp}(q_n, T_n, b), \ldots)) \), then \( \{T\}_q = \{T_1\}_{q_1}, \ldots, \{T\}_{q_n} = \{T_n\}_{q_n} \).

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 8.8 (Completion).** Given a CT-unification problem \( T_L \equiv^? T_R \) and the corresponding qualified unification problem \( T'_L \equiv^? T'_R \), if \( \sigma_Q \) is a unifier for \( T'_L \equiv^? T'_R \) and \( \sigma_U \) is the unifier attained by completing \( \sigma_Q \), then for any super-complete decision \( \mathfrak{S} \), \( [T_L \sigma_U]_{\mathfrak{S}} \equiv [T'_L \sigma_Q]_{\mathfrak{S}} \) and \( [T_R \sigma_U]_{\mathfrak{S}} \equiv [T'_R \sigma_Q]_{\mathfrak{S}} \).

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
Completion also preserves principality since the \( \text{comp} \) 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 8.8, stating that through qualification and completion, we can solve CT-unification problems. We call this process variational unification.

**Theorem 8.9.** Given a CT-unification problem \( U \) and the corresponding qualified unification problem \( Q \), if \( \sigma_Q \) is a unifier for \( Q \), then we can attain a unifier \( \sigma_U \) 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.

### 8.4. Time Complexity of the Unification Algorithm

Solving the unification problem \( U \) consists of three steps: transforming \( U \) into the corresponding qualified unification problem \( Q \), solving \( Q \) with the qualified unification algorithm \( \text{unify} \), and transforming the qualified unifier into the variational unifier using the completion process \( \text{comp} \). 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 \( \sigma_U \) and \( \sigma_Q \) to denote the unifier for \( U \) and \( Q \), respectively. The size of a type is the number of nodes in its AST (as defined by the grammar in Section 4.1). The size of \( T_L \) is assumed to be \( l \) and the size of \( T_R \) is assumed to be \( r \).

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

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 that \( \text{choices}(T_L) \cap \text{choices}(T_R) = \emptyset \). 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 sub-problems with \( l \) nodes in each LHS and a total of \( r - 1 \) nodes between the two RHSs. The split-hoist-decompose process will be recursively applied to each sub-problem until the RHS of all sub-problems is either a type variable or a ground plain type. This will take a total of \( (r - 1)/2 \) applications of the process, equal to the number of internal nodes in \( T_R \). Thus, the overall time complexity of solving \( Q \) is \( O(lr) \).

Finally, we consider the complexity of the third step of the unification process, transforming the solution \( \sigma_Q \) for \( Q \) into a solution \( \sigma_U \) for \( U \) through the process of completion. Again, we perform a worst-case analysis. Completion is performed by folding the mappings in \( \sigma_Q \) with the function \( \text{comp} \). We can establish an upper bound on the number of mappings in \( \sigma_Q \) 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 \( (r - 1)/2 \) sub-problems of the form \( T'_L \equiv T \). 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, \( \sigma_Q \) 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 \( a_{q_i} \rightarrow T_i \in \sigma_Q \) essentially describes a leaf
in that tree. Applying \textit{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 \( O(l^2r) \).

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.

9. 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 \( \text{infer} \) and its type is given below.

\[
\text{infer} : \Delta \times \Lambda \times \Gamma \times e \rightarrow \sigma \times T
\]

The function takes four arguments: the three environments maintained in the typing rules (the dimension environment \( \Delta \), the \textit{share}-bound variable environment \( \Lambda \), and the type environment \( \Gamma \)) and the expression to type. It returns a type substitution and the inferred type.

The cases of the \( \text{infer} \) algorithm can be derived from the typing rules in Section 4.2. The cases for dimensions, choices, and application are given below.

\[
\text{infer}(\Delta, \Lambda, \Gamma, \text{dim } D(t_1, t_2) \text{ in } e) = \begin{cases} 
\text{return } \text{infer}(\Delta \oplus (D, D'), \Lambda, \Gamma, e) & \text{\{\( D' \) is a fresh dimension name\}} \\
\text{\} \end{cases}
\]

\[
\text{infer}(\Delta, \Lambda, \Gamma, D(e_1, e_2)) = D' = \Delta(D) \\
(\sigma_1, T_1) = \text{infer}(\Delta, \Lambda, \Gamma, e_1) \\
(\sigma_2, T_2) = \text{infer}(\Delta, \Lambda, \Gamma \sigma_1, e_2) \\
\text{return } (\sigma_2 \circ \sigma_1, D'(T_1, T_2))
\]

\[
\text{infer}(\Delta, \Lambda, \Gamma, e_1 e_2) = \\
(\sigma_1, T_1) = \text{infer}(\Delta, \Lambda, \Gamma, e_1) \\
(\sigma_2, T_2) = \text{infer}(\Delta, \Lambda, \Gamma \sigma_1, e_2) \\
\sigma = \text{unify}'(T_3 \sigma_2, T_2 - a) & \text{\{\( a \) is a fresh type variable\}}
\]

return \( (\sigma \circ \sigma_2 \circ \sigma_1, a)\)

The case for dimension declarations is the simplest shown. It simply returns the inferred type and substitution of its body after adding a new mapping to the dimension environment, as described in Section 4.2.

On a choice, we lookup this mapping to determine the dimension name of the resulting choice type. 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 resulting types and mappings will be consistent. Finally, the resulting mapping is just a composition of the mappings produced during type inference of the two alternatives.

The \( \text{infer} \) 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 \( \text{unify}' \) 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 $\sigma$, the solution to the original unqualified unification problem. The remaining cases are similarly straightforward. Abstractions and $\lambda$-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 dimension and choice cases above.

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

**Theorem 9.1 (Type Inference is Sound).** \[ \text{infer}(\Delta, \Lambda, \Gamma, e) = (\sigma, T) \implies \Delta, \Lambda \sigma \vdash e : T. \]

**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 8.9, this property is preserved.

The type inference algorithm also has the principal typing property, which follows from Theorems 8.6 and 8.9.

**Theorem 9.2 (Type Inference is Complete and Principal).** For every mapping $\sigma$ and type $T$ such that $\Delta, \Lambda, \Gamma \sigma \vdash e : T$, there exists a $\sigma'$ and $T'$ such that infer$(\Gamma, e) = (\sigma', T')$ where $\sigma = \sigma'' \circ \sigma'$ for some $\sigma''$ and $T = T' \sigma'''$ for some $\sigma'''$.

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

**10. 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.

**10.1. 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.

\[ \dim A \langle t_1, t_2 \rangle \text{ in } \dim B \langle u_1, u_2 \rangle \text{ in } A \langle B \langle e_1, e_2 \rangle, B \langle e_3, e_4 \rangle \rangle \]
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 $D(e_1, \ldots, e_n)$ 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 $C[D(e_1, \ldots, e_n)]$. 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, $A(e_1, \ldots, e_n) B(e'_1, \ldots, e'_m)$. 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 naming of variables, or only the implementation of a function), this offers a dramatic opportunity for efficiency gains.

10.2. Experimental Demonstration

In this section we continue the efficiency discussion by demonstrating the performance of our variational type inference algorithm, $\text{infer}$, 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. For space reasons, explicit dimension declarations are omitted from the table, but all seemingly free dimensions are actually defined in the immediate context of the expression.
We can also observe that the ratio of overhead for inference is 4. Now we can see the exponential explosion of the brute-force strategy (which must also double the number of dimensions from 2 to 4). While the running time for the brute-force strategy correspondingly scales essentially linearly with respect to the size of the expression. We can also observe that the ratio of overhead for inference, 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, e4. As expected, inference performs worse than brute-force due to the overhead. With e5, however, we demonstrate how even a very small amount of common context can tip the scale back in inference's favor (id^3 denotes three applications of the id function). If we again duplicate the initial expression, as in e6, we introduce an opportunity for sharing, allowing inference to scale nicely while brute-force does not.

Next we analyze the impact of a difficult case for our algorithm that we call long cascading. 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.

<table>
<thead>
<tr>
<th>expression</th>
<th>dims</th>
<th>variants</th>
<th>one</th>
<th>brute</th>
<th>vlc</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1 A(λx.3, λx.true) B(3,5)</td>
<td>2</td>
<td>4</td>
<td>2.6</td>
<td>10.2</td>
<td>10.1</td>
</tr>
<tr>
<td>e2 A(λx.3, λx.true) B(3,5,7,9)</td>
<td>2</td>
<td>8</td>
<td>2.5</td>
<td>20.4</td>
<td>10.2</td>
</tr>
<tr>
<td>e3 if true e1 e1</td>
<td>4</td>
<td>16</td>
<td>21.2</td>
<td>334.7</td>
<td>34.3</td>
</tr>
<tr>
<td>e4 A(B(succ, λx.true),B(λx.3,not)) B(3,true)</td>
<td>2</td>
<td>4</td>
<td>2.6</td>
<td>10.1</td>
<td>15.7</td>
</tr>
<tr>
<td>e5 id^3 e4</td>
<td>2</td>
<td>4</td>
<td>31.9</td>
<td>125.1</td>
<td>63.0</td>
</tr>
<tr>
<td>e6 if true e4 e4</td>
<td>4</td>
<td>16</td>
<td>24.0</td>
<td>380.5</td>
<td>55.0</td>
</tr>
</tbody>
</table>

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 dims column indicates the number of independent 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 inference 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 Bool → a → a → a.

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 e1, 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 inference 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 e2 we have doubled the number of variants by increasing the number of tags 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 e3 we duplicate e1 and add some additional, unvaried context (if True). This doubles the number of dimensions from e1 and increases the number of variants by a factor of 4. Now we can see the exponential explosion of the brute-force strategy (which must also type the common context 16 times), while inference scales essentially linearly with respect to the size of the expression. We can also observe that the ratio of overhead for inference, 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, e4. As expected, inference performs worse than brute-force due to the overhead. With e5, however, we demonstrate how even a very small amount of common context can tip the scale back in inference’s favor (id^3 denotes three applications of the id function). If we again duplicate the initial expression, as in e6, we introduce an opportunity for sharing, allowing inference to scale nicely while brute-force does not.

ACM Transactions on Programming Languages and Systems, Vol. V, No. N, Article A, Publication date: January YYYY.
Fig. 14. Performance comparison for the long cascading problem.

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

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

Figure 14 demonstrates the performance of `infer` and the brute-force algorithm on expressions designed to induce the long cascading 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 long cascading 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) 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. The table gives the size of each expression as the number of AST nodes and the number of contained (binary) dimensions. All dimensions are at the top level, so an expression with d dimension 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).
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 that the longest occurrences of long cascading, 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 variants using the brute-force strategy, and to infer a variational type using \texttt{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.

11. RELATED WORK

A limitation of the type system presented in this paper is that a VLC expression is typable only if all of its variants are well typed. This is not very robust since 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 detail 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.

Other research related to the representation of variational programs was covered in depth in Section 2. In this section we address work related to the concept of variational types, VLC’s type system, typing variational programs, and the variational type inference algorithm. This work falls roughly into two categories. In Section 11.1 we discuss related theoretical work in the area of programming languages and type systems. In Section 11.2 we relate our approach to the often more pragmatic work done in the area of software product lines.

11.1. 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 \( \text{Int} \) and \( \text{Bool} \). 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 \( \text{Int} \) or type \( \text{Bool} \). 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] propose 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, \( \langle A(a, b), c \rangle \equiv \langle a, c \rangle \) but \( \{a, b, c\} \neq \{a, c\} \).

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 has 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 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.

11.2. Type Checking Software Product Lines

In the context of software product lines (SPLs), some 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 develop-
opment [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 these products is achieved in two steps. In the first step, 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.

Also in the field of SPLs, Kästner et al. [2012] describe a type system for Colored Featherweight Java (CFJ) [Apel et al. 2008]. 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, but they differ in the kinds of variations that are supported. Specifically, VLC supports alternative variation (which subsumes optional variation as a special case), whereas CFJ supports only optional variation. 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 types.

Finally, there has 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 symbols 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. 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.

12. 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 $W$.

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 typing 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 other extensions of the type system and their interaction with choice types. We will also apply the method demonstrated in this paper to other languages.

REFERENCES


A.36

Sheng Chen et al.


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 8.1 and discussed in Section 8.1. 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 8.6 and discussed in Section 8.3. 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 8.1.** Given a CT-unification problem \( U \), there is a unifier \( \sigma \) such that for any unifier \( \sigma' \), there exists a mapping \( \theta \) 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 \( \text{vars}(U) \) 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.

\[
[T_L \equiv^? T_R]_s = [T_L]_s \equiv^? [T_R]_s
\]

\[
[\sigma]_s = \{(a, [T]_s) \mid (a, T) \in \sigma\}
\]

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

**Lemma A.1 (Selection extends over substitution).** \( [T\sigma]_s = [T]_s[\sigma]_s \)

**Proof of Lemma A.1.** The proof is based on induction over the structure of \( T \) and \( \sigma \). 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 \( \sigma \).

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

\[
[T\sigma]_s = [D(T_1, T_2)\sigma]_\tilde{D}
\]

\[
= [D(T_1\sigma, T_2\sigma)]_\tilde{D}
\]

\[
= [T_2\sigma]_\tilde{D}
\]

\[
= [T_2]_\tilde{D}[\sigma]_\tilde{D}
\]

\[
= [D(T_1, T_2)]_\tilde{D}[\sigma]_\tilde{D}
\]

\[
= [T]_s[\sigma]_s
\]
(2) Given \( T = a \), assume \( a \sigma = D(T_1, T_2) \) and \( s = \bar{D} \) (again \( s = D \) is dual).

\[
[T]_s [\sigma]_s = [a]_D [\sigma]_D = a \sigma'[a = [T_2]_D] \quad \text{by definition}
\]

\[
[T]_s [\sigma]_s = [a \sigma]_D = [D(T_1, T_2)]_D \quad \text{by definition}
\]

The remaining cases can be constructed similarly. \( \square \)

From Lemma A.1, it follows by induction that the same result holds for decisions as for single selectors: \([T \sigma]_T = [T]_T [\sigma]_T\). Combining this with Lemma 5.2 (selection preserves type equivalency), we see that if \( \sigma \) is a unifier for \( U \) then \([T_L \sigma]_T \equiv [T_R \sigma]_T\) for any \( \bar{s} \). This is the same as saying that if \( T_L \sigma \neq T_R \sigma \), then \( \exists \bar{s} : [T_L \sigma]_T \neq [T_R \sigma]_T\). A direct consequence of this result is that if \( \bar{s} \) is super-complete (it eliminates all choice types in \( T_L, T_R \), and \( \sigma \)) and \([T_L]_T [\sigma]_T \equiv [T_R]_T [\sigma]_T\), then \( \sigma \) is a unifier for \( U \).

**Proof of Theorem 8.1.** Using the type splitting algorithm described in Section 8.2, we can transform \( U \) into \( Q \) such that for all super-complete decisions \( \bar{s}_1, \bar{s}_2, \ldots, \bar{s}_n \), if \( \bar{s}_i \neq \bar{s}_j \), then \( \text{vars}([Q]_{\bar{s}_i}) \cap \text{vars}([Q]_{\bar{s}_j}) = \emptyset \).

Each subproblem \([Q]_{\bar{s}_i}\) corresponding to a super-complete decision \( \bar{s}_i \) is plain. Therefore, we can obtain (via Robinson’s algorithm) an mgu \( \sigma_i \) such that \([T_L^i \sigma_i]_{\bar{s}_i} \equiv [T_R^i \sigma_i]_{\bar{s}_i}\). Let \( \sigma \) be the disjoint union of all of these mgus, \( \sigma = \bigcup_{i \in \{1, \ldots, n\}} \sigma_i \).

Since the type variables in each subproblem are different, for each subproblem we have \([T_L^i \sigma_i]_{\bar{s}_i} \equiv [T_R^i \sigma_i]_{\bar{s}_i}\). Then based on the discussion after Lemma A.1, \( \sigma \) is a unifier for \( T_L \equiv T_R \). Moreover, it is most general by construction since each \( \sigma_i \) is most general. Based on Theorem 8.9 (variational unification is sound) and Lemma 8.7 (comp is correct and preserves principality), the completion of \( \sigma \) is the mgu for \( U \), which proves that variational unification is unitary. \( \square \)

### 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 8.3.

#### A.2.1. Proof of Theorem 8.5

**Theorem 8.5 (Soundness).** If \( \text{unify}(T_1, T_2) = \sigma \), then \( T_1 \sigma = T_2 \sigma \).

**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 \( \text{unify} \) 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 \( \text{unify} \) 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 \( a_q \equiv D(a_p q, \bar{a}_p) \).

3. Two choice types in the same dimension. Decomposition by alternatives is correct by the inductive hypothesis and Lemma 8.2.

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 8.3.
(c) Splitting is choice independent by Lemma 8.4.

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 \( (T_1 \rightarrow T_2)\sigma = (T'_1 \rightarrow T'_2)\sigma \) by an application of the \texttt{FUN} equivalence rule in Figure 5.

7. The last case considers a qualified type variable \( a_q \) and a function type \( T \rightarrow T' \). 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. \( \square \)

### A.2.2. Proof of Lemma 8.7

**Lemma 8.7.** Given a mapping \( \{a_q \rightarrow T'\} \), if \( T = \text{comp}(q,T',b) \) is the completed type (where \( b \) is fresh), then \( [T]_q = [T']_q \). More generally, given \( \{a_q \rightarrow T_1, \ldots, a_{q_n} \rightarrow T_n\} \), if \( T = \text{comp}(q_1,T_1,\text{comp}(q_2,T_2, \ldots, \text{comp}(q_n,T_n,b)) \ldots) \), then \( [T]_{q_1} = [T]_{q_1} \ldots, [T]_{q_n} = [T]_{q_n} \).

**Proof.** We can prove the first part of this theorem by structural induction on the qualifier \( q \). The base case, where \( q = \) the empty qualifier \( \epsilon \), is trivial since \( \text{comp}(\epsilon,T',b) = T' \). We show the inductive case below for \( q = D_q \) (the case for \( q = D_q' \) is dual). Note that the induction hypothesis is \( \{\text{comp}(q',T',b)\}_q' = [T']_q' \).

\[
\begin{align*}
[T]_q &= [\text{comp}(D_q,T',b)]_{D_q'} & \text{by assumption} \\
&= [D(\text{comp}(q',T',b),\text{fresh}(b))]_{D_q'} & \text{definition of comp} \\
&= [\text{comp}(q',T',b)]_q' & \text{definition of repeated tag selection} \\
&= [\text{comp}(q',T',b)]_q'_{D_q'} & \text{selector ordering is irrelevant} \\
&= [T']_q'_{D_q} & \text{induction hypothesis} \\
&= [T']_q & \text{selector ordering is irrelevant}
\end{align*}
\]

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 \( \text{comp} \) is commutative, for example, \( \text{comp}(q_1,T_1,\text{comp}(q_2,T_2,b)) = \text{comp}(q_2,T_2,\text{comp}(q_1,T_1,b)) \). \( \square \)

### A.2.3. Proof sketch of Lemma 8.8

**Lemma 8.8 (Conclusion).** Given a CT-unification problem \( T_L = T_R \) and the corresponding qualified unification problem \( T'_L = T'_R \), if \( \sigma_Q \) is a unifier for \( T'_L = T'_R \), \( \sigma_U \) is the unifier attained by completing \( \sigma_Q \), then for any super-complete decision \( \tilde{s} \), \( [T_L]_{\tilde{s} \sigma_U} = [T'_L]_{\tilde{s} \sigma_Q} \) and \( [T_R]_{\tilde{s} \sigma_U} = [T_R]_{\tilde{s} \sigma_Q} \).

**Proof Sketch.** The proof is based on induction over the structure of the types \( T_L \) and \( T_R \), the super-complete decision \( \tilde{s} \), and the unifier \( \sigma_Q \). 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, \( [T_L]_{\tilde{s} \sigma_U} = [T_a]_{\tilde{s} \sigma_Q} \).
   (b) \( T_L = T_{\text{arg}} \rightarrow T_{\text{res}} \). The induction hypothesis is that \( [T_{\text{arg}}]_{\tilde{s} \sigma_U} = [T_{\text{arg}} \sigma_Q]_{\tilde{s} \sigma_Q} \) and \( [T_{\text{res}}]_{\tilde{s} \sigma_U} = [T_{\text{res}} \sigma_Q]_{\tilde{s} \sigma_Q} \). Then \( ([T_{\text{arg}} \rightarrow T_{\text{res}} \sigma_U]_{\tilde{s} \sigma_Q} = [T_{\text{arg}} \sigma_Q]_{\tilde{s} \sigma_Q} \rightarrow [T_{\text{res}} \sigma_Q]_{\tilde{s} \sigma_Q} \rightarrow \cdots = [T_{\text{arg}} \rightarrow T_{\text{res}} \sigma_Q]_{\tilde{s} \sigma_Q} \).
A.3. Other Collected Proofs

A.3.1. Proof sketch of Lemma 5.2

Lemma 5.2 (Type equivalence preservation). If $T_1 \equiv T_2$, then $[T_1]_{D[1]} = [T_2]_{D[1]}$.

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 name matches that of the selector, or it does not. We consider the sub-case where the dimension name does not match first.

$T_L = D(T_1, T_2), \bar{s} = \bar{s}_1$. Let $T'_L = D(T'_1, T'_2)$, the induction hypothesis is that $[T_1\sigma_{U}]_{D[1]} = [T'_1\sigma_{U}]_{D[1]}$ and $[T_2\sigma_{U}]_{D[1]} = [T'_2\sigma_{U}]_{D[1]}$ for any $\bar{s}_1$.

$$[T_L\sigma_{U}]_{D[1]} = [D(T_1, T_2)\sigma_{U}]_{D[1]} = \frac{[T_1\sigma_{U}]_{D[1]} + [T_2\sigma_{U}]_{D[1]}}{\bar{s}_1}$$

by assumption

$$= [T_2\sigma_{U}]_{D[1]}$$

tag selection

$$= [T_2\sigma_{U}]_{D[1]}$$

by Lemma A.1

$$= [T_2\sigma_{U}]_{D[1]}$$

selector ordering is irrelevant

$$= [T'_2\sigma_{U}]_{D[1]}$$

induction hypothesis

$$[T'_L\sigma_{U}]_{D[1]} = \ldots = [T'_2\sigma_{U}]_{D[1]}$$

The cases for $T_R$ are dual. □

A.3. Other Collected Proofs

A.3.1. Proof sketch of Lemma 5.2

$T_L = D(a, T_2), \bar{s} = D$. In this case, $T'_L = D(a_D, T'_2)$.

$$[T_L\sigma_{U}]_{D[1]} = [D(a, T_2)\sigma_{U}]_{D[1]} = \frac{[a\sigma_{U}]_D + [D(a_D)\sigma_{U}]_{D[1]}}{\bar{s}_1}$$

by assumption

$$= [a\sigma_{U}]_D$$

tag selection and Lemma A.1

$$= [a\sigma_{U}]_D$$

by Lemma A.1

$$= [D(a_D, T'_2)\sigma_{U}]_{D[1]}$$

tag selection

$$= [T'_L\sigma_{U}]_{D[1]}$$

The cases for $T_R$ are dual. □

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 $s = D$ (selecting the second alternatives). The case for $s = D$ is dual to this.

$$[D(T_1, T_2)\sigma_{U}]_{D[1]} = [D(T_1, T_2)\sigma_{U}]_{D[1]} = \frac{[T_2\sigma_{U}]_D + [D(T_2)\sigma_{U}]_{D[1]}}{\bar{s}_1}$$

selection in LHS

$$\equiv D(T_1, T_2)\sigma_{U} = D(T_1, T_2)\sigma_{U}$$

by definition

$$[D(T_1, T_2)\sigma_{U}]_{D[1]} = [D(T_1, T_2)\sigma_{U}]_{D[1]} = \frac{[T_2\sigma_{U}]_D + [D(T_2)\sigma_{U}]_{D[1]}}{\bar{s}_1}$$

selection in RHS

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 $s = D$ (selecting the second alternatives). The case for $s = D$ is dual to this.

$$[D(T_1, T_2)\sigma_{U}]_{D[1]} = [D(T_1, T_2)\sigma_{U}]_{D[1]} = \frac{[T_2\sigma_{U}]_D + [D(T_2)\sigma_{U}]_{D[1]}}{\bar{s}_1}$$

selection in LHS

$$\equiv D(T_1, T_2)\sigma_{U} = D(T_1, T_2)\sigma_{U}$$

by definition

$$[D(T_1, T_2)\sigma_{U}]_{D[1]} = [D(T_1, T_2)\sigma_{U}]_{D[1]} = \frac{[T_2\sigma_{U}]_D + [D(T_2)\sigma_{U}]_{D[1]}}{\bar{s}_1}$$

selection in RHS

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 $s$
Extending Type Inference to Variational Programs

makes a selection in \( D \) follows.

\[
[D'(T_1, D(T_2, T_3))]_D = D'([T_1]_D, [D(T_2, T_3)]_D) \quad \text{selection in LHS}
\]

\[
=D'([T_1]_D, [T_3]_D) \quad \text{by definition}
\]

\[
[D(D'(T_1, T_2), D'(T_1, T_3))]_D = [D'(T_1, T_3)]_D \quad \text{selection in RHS}
\]

\[
=D'([T_1]_D, [T_3]_D) \quad \text{by definition}
\]

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.

\[ \square \]

A.3.2. Proof of Lemma 5.4

**Lemma 5.4 (Local Confluence).** For any type \( T \), if \( T \leadsto T_1 \) and \( T \leadsto T_2 \), then there exists some type \( T' \) such that \( T_1 \leadsto T' \) and \( T_2 \leadsto T' \).

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 \( \epsilon \). We use \( T|_p \) to refer to the type at position \( p \) in type \( T \). For example, given \( T = \text{Int} \rightarrow A(\text{Bool}, \text{Int}) \), we can refer to the component types of \( T \) in the following way.

\[
T|_\epsilon = \text{Int} \rightarrow A(\text{Bool}, \text{Int})
\]

\[
T|_L = \text{Int}
\]

\[
T|_R = A(\text{Bool}, \text{Int})
\]

\[
T|_{RL} = \text{Bool}
\]

\[
T|_{RR} = \text{Int}
\]

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, \( T[\text{Bool}]|_R = \text{Int} \rightarrow \text{Bool} \). We use \( \mathcal{P}(T) \) 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 \( l \leadsto r \) 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 \( \sigma \) 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 \( T = \text{Int} \rightarrow A(\text{Bool}, \text{Bool}) \), we can apply the S-C-IDEMP rule \( (l = A(x, x), r = x) \) at \( p = R \) with the substitution \( \sigma = [x \mapsto \text{Bool}] \), resulting in \( T' = \text{Int} \rightarrow \text{Bool} \) (note that we assume the dimension name in the simplification rule is instantiated automatically).

**Proof.** Given type \( T \), assume that some rewrite rule \( l_1 \leadsto r_1 \) can be applied at position \( p_1 \) with substitution \( \sigma_1 \), and another rewrite rule \( l_2 \leadsto r_2 \) can be applied at position \( p_2 \) with substitution \( \sigma_2 \). 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 \( T_1 \leadsto T' \) and \( T_2 \leadsto T' \). 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 \( l_1 \leadsto r_1 \) first, we can reach \( T'' \) by next applying \( l_2 \leadsto r_2 \), 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 $\sigma_1$ (the case where $p_2$ is a prefix of $p_1$ is dual). For example, consider the following type $T$.

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

We can apply the S-C-Swap1 rule, $A\langle B\langle x, y, z \rangle \rangle \leadsto A\langle B\langle x, z, B\langle y, z \rangle \rangle \rangle$, at position $p_1 = \epsilon$ with the substitution $\sigma_1 = \{ x \leftarrow Int, y \leftarrow Bool, z \leftarrow C\langle Int, Int \rangle \}$. Or we can apply the S-C-Idemp rule, $C\langle w, w \rangle \leadsto w$, at position $p_2 = R$ with the substitution $\sigma_2 = \{ w \leftarrow Int \}$. Note that $l_2\sigma_2 = D\langle Int, Int \rangle$, which is in the range of $\sigma_1$. 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 need to apply the $l_2 \leadsto r_2$ 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 $p \in \mathcal{P}(l_1)$ 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 $T$.

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

Then both S-C-Swap 1 and S-C-Swap 2 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 $p \in \mathcal{P}(l_1)$ such that $l_1|p$ is not a type variable, compute the mgu for $l_1|p \equiv \? l_2$ as $\theta$, then $r_1\theta$ and $l_1|\theta|\theta|\theta|\theta$ 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.)

\begin{align*}
\text{S-C-Swap1} \quad & C(A(x, y), z) \rightarrow A(C(x, z), C(y, z)) \\
\text{S-C-Swap2} \quad & C(l, B(m, n)) \rightarrow B(C(l, m), C(l, n))
\end{align*}

When \( p = c \), the unification problem is \( C(A(x, y), z) \equiv C(l, B(m, n)) \). The computed \( \text{mgu} \) is given below, where all previously undefined type variables are fresh.

\[
\theta = \{ x \mapsto C(l, b), y \mapsto C(d, l), e \mapsto C(f, B(m, n)) \}
\]

The critical pair consists of the following two types.
1. \( A(C(A(l, b), c), C(f, B(m, n))) \), \( C(C(A(l, d), e), C(f, B(m, n))) \)
2. \( B(C(l, m), C(l, n)) \)

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

\[
\begin{align*}
A(C(A(l, b), c), C(f, B(m, n))) &= A(C(l, B(m, n)), C(l, B(m, n))) & \text{S-C-Dom1 and S-C-Dom2} \\
C(C(A(l, d), e), C(f, B(m, n))) &= C(l, B(m, n)) & \text{S-C-Idemp} \\
B(C(l, m), C(l, n)) &= B(C(l, m), C(l, n)) & \text{S-C-Swap2}
\end{align*}
\]

When \( p = L \), the unification problem is \( A(x, y) \equiv C(l, B(m, n)) \), and the computed \( \text{mgu} \) is given below.

\[
\theta = \{ x \mapsto A(C(l, B(m, n), a), y \mapsto A(b, C(l, B(m, n))) \}
\]

The critical pair consists of the following two types.
1. \( A(C(l, B(m, n), a), z), C(A(b, C(l, B(m, n))), z) \)
2. \( C(B(l, m), C(l, n)), z) \)

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

\[
\begin{align*}
A(C(A(l, B(m, n), a)), z), C(A(b, C(l, B(m, n))), z) &= A(C(l, z), C(l, z)) & \text{S-C-Dom1 and S-C-Dom2} \\
\text{S-C-Dom2} \quad & C(l, z) & \text{S-C-Idemp}
\end{align*}
\]

\[
\begin{align*}
C(B(l, m), C(l, n)), z) &= C(B(l, l), z) & \text{S-C-Dom1 and S-C-Dom2} \\
\text{S-C-Dom2} \quad & C(l, z) & \text{S-C-Idemp}
\end{align*}
\]

The proofs for other critically overlapping rules can be constructed similarly. \( \Box \)