An Error-Tolerant Type System for Variational Lambda Calculus

Sheng Chen, Martin Erwig, Eric Walkingshaw
School of EECS, Oregon State University
{chensh,erwig,walkiner}@eecs.oregonstate.edu

Abstract

Conditional compilation and software product line technologies make it possible to generate a huge number of different programs from a single software project. Typing each of these programs individually is usually impossible due to the sheer number of possible variants. Our previous work has addressed this problem with a type system for variational lambda calculus (VLC), an extension of lambda calculus with basic constructs for introducing and organizing variation. Although our type inference algorithm is more efficient than the brute-force strategy of inferring the types of each variant individually, it is less robust since type inference will fail for the entire variational expression if any one variant contains a type error. In this work, we extend our type system to operate on VLC expressions containing type errors. This extension directly supports locating ill-typed variants and the incremental development of variational programs. It also has many subtle implications for the unification of variational types. We show that our extended type system possesses a principal typing property and that the underlying unification problem is unitary. Our unification algorithm computes partial unifiers that lead to result types that (1) contain errors in as few variants as possible and (2) are most general. Finally, we perform an empirical evaluation to determine the overhead of this extension compared to our previous work, to demonstrate the improvements over the brute-force approach, and to explore the effects of various error distributions on the inference process.

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

Keywords error-tolerant type systems; variational lambda calculus; variational type inference; variational types

1. Introduction

The source code of many software projects can be used to generate a huge number of distinct programs that run on different platforms and provide different sets of features. Current research on software product lines (SPLs) [20] and feature-oriented software development [2] provide processes and tools for the development of massively configurable software, suggesting that the variability of software systems will only continue to grow. Unfortunately, basic program verification tools, such as type systems, are not equipped to deal with variation on this scale. Notions of type correctness are defined in terms of single programs only, but generating all program variants and testing each one individually is usually impossible due to the sheer number of variants that can be generated.

The problem of type checking variational software is an active area of research [11, 12, 25]. Most of this work comes out of the SPL community and is therefore highly pragmatic, tool-oriented, and focused on imperative languages. Our work on this problem, begun in [5], distinguishes itself in several ways. Most significantly, while other approaches consider only type checking of programs in explicitly typed languages, we solve the more general problem of *type inference* for implicitly typed languages. Our approach begins by establishing a simple functional language, the *variational lambda calculus* (VLC), for studying variational software; it introduces a notion of *variational types* for typing variational programs; it develops a formal type system that associates variational types with VLC expressions; and it presents an algorithm that infers these types. By addressing the problem from a more theoretical and fundamental perspective, we believe our results are more reusable and extensible than others. Variational types are also a general contribution to type theory that have other potential applications; for example, they may be useful for more flexibly typing metaprograms.

A subtle difference between the problems of checking explicitly typed programs and inferring types in implicitly typed programs is that, in general, a type error encountered during inference prevents inference in the rest of the program. This means that while our solution is more general, it is less *robust*. A type error in a single variant will cause the entire inference process to fail. In this work we extend our type system and inference algorithm to allow for type errors at arbitrary positions in the inferred variational type. This extension directly supports the location of ill-typed variants, and the ability to incrementally develop variational programs by leaving some variational branches undefined or incomplete while other variants are extended and fleshed out. While the focus in [5] is on establishing a broad foundation for formal work on typing and other static analyses of variational programs, here we focus on solving a specific problem of practical importance. Solving this problem is surprisingly challenging and leads to many interesting theoretical results, summarized in Section 1.2.

1.1 Motivation

In this section we will briefly motivate this work by way of a simple example. We also motivate and explain our choice of VLC as a formal foundation for typing variational programs.

In general, there are three competing approaches to managing variational software, each with their own strengths and weaknesses. *Compositional* approaches rely on language features like mixins [4] or aspects [16] to modularize features that may or may not be included in a generated variant. This approach is mostly used in conjunction with object-oriented programming languages. *Metaprogramming*-based approaches rely on staged computations to generate program variants through the use of macros; this is especially

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

common in functional languages, for example, MetaML [24] and the Lisp family. Finally, *annotative* approaches rely on a separate annotation language to embed static variation directly within the source code. The C Preprocessor (CPP) is by far the most widely used annotative variation tool. One of the advantages of the annotative approach is that it is mostly independent of the object language and so can be applied across paradigms (and even in documentation and other non-source code). CPP annotations are frequently seen in large-scale Haskell programs, for example, GHC [9].

Although all three approaches are worthy of study, we choose the annotative approach here because it makes the variation in a program explicit, allowing us to directly traverse and manipulate the variation structure. This is not the case, for example, in metaprogramming approaches, where variability is captured only implicitly in the definition and use of macros. While our annotation language is much less powerful than metaprogramming systems, it allows us to support a much more general form of type-safe variation than is possible in, for example, MetaML.

Consider two different ways to implement a function in Haskell to find values in a lookup list of type `[(a,b)]`. In the first, we return a value of type `Maybe b`, possibly containing the first value in the lookup list associated with a given key of type `a`.

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

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

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

Based on a notation developed in [8], we can represent the variation between these two function implementations by annotating the program in-place. First, we declare a new *dimension* of variation, *Res*, representing variation in the function's result. Then we indicate the specific variation points in the code using *choices* that are bound to the *Res* dimension.

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

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

The types inferred in a variational program are also variational. For our `find` function, we infer the following *variational type* which also contains a choice in the *Res* dimension.[1]

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

Using the variational type inference algorithm we have developed in [5] we can infer types like the above. A successfully inferred variational type indicates that all variants of the program are type correct. Since the typing information of shared code is reused (and for other reasons), a type-correctness result can be obtained much more efficiently in the expected case than the brute-force strategy of generating all variants and type checking them separately. For large variational programs with many dimensions of variation, the efficiency gains can make type checking all variants tractable, when otherwise it would not be.

However, variational type inference has a hidden cost relative to the brute-force strategy. While variational type inference is more efficient at *detecting errors*, it is less useful for *locating errors*. To demonstrate, suppose we add a new dimension of variation to our `find` function, *Arg*, that captures variation between looking up values based on an example key (as above) or looking up values based on a predicate on keys. We name the tags corresponding to these possibilities *val* and *pred*, respectively.

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

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

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

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

This can be easily fixed by replacing `x` with the choice $Arg\langle x, p\rangle$. The problem is that the type inference algorithm presented in [5] provides no hint at the location of this error—it just fails, indicating that there *is* an error. The brute-force strategy is more robust. By type checking each variant individually, we can determine exactly which variant(s) contain type errors and infer types for those that are type correct. Of course, the brute-force strategy scales just as poorly for error location as it does for type checking (although it might be able to be used strategically, if one can correctly guess the variants that contain errors).

In this paper we extend variational type inference to return partially correct variational types—that is, variational types containing errors. For example, the errorful variational type of our `find` function can be written as follows, where $\bot$ is a special type that indicates a type error at that location in the type.

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

This type indicates that there is a type error in the result type of the function if the second tag is chosen from each dimension (*Arg.pred* and *Res.all*). This extension therefore directly supports the location of type errors in variational programs without resorting to the brute-force strategy of typing variants individually. Similarly, it supports type inference on incomplete variational programs—programs in which only some variants are in a complete and type-correct state—a quality which is needed for incremental development.

The addition of error types is a non-trivial extension to the type system and inference algorithm presented in [5]. In particular, there are many subtle implications for the unification of variational types. In the case of an unbound variable, as above, the location of the error is obvious. However, often there are many possible candidates for the type error, depending on how we infer the surrounding types. The goal is to assign errors such that as few variants as possible are considered ill-typed, that is, to find a type that is *most-defined*. This goal is in addition to the usual goal of inferring the *most general* type possible. It is not obvious whether these two qualities of types are orthogonal. In this paper we will show that they are, and we present an inference algorithm that identifies most-defined, most-general types.

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

1.2 Contributions and Rest of Paper

In the next section we briefly introduce the syntax and semantics of VLC, developed in [5], which is the formal foundation of this work. The structure of the rest of the paper is described relative to the major contributions of this work, which are:

1. The extension of our variational type system to support the typing of programs in which not all variants are well typed. The extension of the types themselves is discussed in Section 3, and the extension of the typing rules in Section 5. A type preservation theorem (Theorem 1) in Section 5 formally establishes the relationship between a variational type identified by our type system and the set of types or type errors produced by the brute-force force approach.

2. The concept of *typing patterns*, defined in Section 4, that indicate which variants of a variational program are well-typed, and an associated *more-defined* relation for comparing them. We use these in Section 6 to prove several results about the problem of unifying variational types containing type errors. Most significantly, we show that for any unification problem, there is a mapping that produces the most-defined result type (Theorem 2), and that among such mappings, there is a unique mapping that produces the most-general result type (Theorem 3).

3. A unification algorithm on variational types with type errors, given in Section 7, that produces unifiers that result in most-defined, most-general types. This is the core component of a type inference algorithm that implements the type system presented in this paper, given in Section 8. We show that both algorithms are sound (Theorems 4 and 6) and complete (Theorem 5 and 7).

4. A theoretical and experimental evaluation of these algorithms. In Section 7, we show that unification of variational types with errors does not increase the complexity of unifying variational types. In Section 9, we conduct experiments that demonstrate that the overhead to support error-tolerant type inference is minor and that our algorithm offers significant performance improvements over the brute-force approach. The evaluation results also reveal an interesting relationship between the distribution of type errors in an expression and the time it takes to infer a type for that expression.

Finally, in Section 10 we discuss related work and offer conclusions and directions for future work in Section 11.

The following table provides a short overview of the notation used throughout the paper. It is meant as an aid to find definitions faster (§ indicates the section(s) containing the definition).

Syntactic Categories	§	Operations	§
Expressions (e)	2.1	Selection $\lfloor e \rfloor_{D.t}$, $\lfloor T \rfloor_{D.i}$	2.2, 3
Types (T)	3	Semantics $[\![\cdot]\!]$	2.2
Typing patterns (P)	4	Masking $P \lhd T$	4
Environments (Γ, Δ)	5	Pattern union $P_1 \oplus P_2$	4
Mappings (θ)	6.2	Type matching $T_1 \bowtie T_2$	4
Partial unifiers (η)	6.2	Arrow lifting $\uparrow(T)$	5
Qual. type vars ($a_{A\bar{B}}$)	7.1	Decision to selectors $\varphi_e(\bar{q})$	5

Relationships	§	Results	§
Equivalence $T_1 \equiv T_2$	3	Type preservation	5
Definedness $P_1 \trianglelefteq P_2$	4	Principal patterns	6.2
More general $\theta_1 \sqsubseteq \theta_2$	6.1	*unify* sound & complete	7.2
		infer sound & complete	8

2. Variational Lambda Calculus

While the example from the previous section was presented in Haskell, here and in our previous work on typing variational functional programs we consider a simpler language, the *variational lambda calculus* (VLC). VLC is a conservative extension of lambda calculus with constructs for introducing and organizing static variation. Constraining the problem to VLC allows us to focus on the fundamental problem of typing variational programs and to present our solution as clearly and simply as possible. In [5] we describe how the variational type system can be extended to incorporate other, more advanced language features. In this section we briefly describe the syntax and semantics of VLC.

2.1 Syntax

VLC is based on our previous work on the *choice calculus* [8]. The choice calculus is a fundamental representation of variation in arbitrary tree structures (such as a program's abstract syntax tree), designed to serve as a general foundation for theoretical research in the field of variation management. The key features of the choice calculus were already introduced in the previous section, namely, *choices* and *dimensions*.[2] Choices specify a point of variation in a tree, while dimensions are used to synchronize and scope related choices.

The syntax of VLC is given below. The first four constructs in the syntax definition correspond to lambda calculus extended with constant values, while the dimension and choice constructs are from the choice calculus. If a VLC expression contains no dimension or choice constructs, we call the expression *plain*.

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

Note that every dimension must contain exactly two tags and all choices must contain exactly two alternatives. This is a constraint made for presentation purposes only. Variation in dimensions with n tags can be easily simulated by $n-1$ binary dimensions. More fundamental syntactic constraints are that the tags associated with one dimension must be different (so that they can be uniquely referred to for selection), and that every choice must occur within scope of a corresponding dimension declaration.

2.2 Semantics

A VLC expression defines a set of *named variants*—a set of plain lambda calculus expressions identified by the selections that must be performed to produce them. These variants are computed *statically*. That is, the full semantics of a VLC expression consists of two distinct stages: a *selection* stage that eliminates all dimensions and choices through tag selection, and an *evaluation* stage that evaluates the resulting plain lambda calculus expression. When we speak of the semantics of a VLC expression in this paper, we refer only to the selection stage, which is briefly described below (a more thorough treatment can be found in [8]).

To select a particular plain expression from a VLC expression, we must repeatedly select tags from dimensions until we are left with an expression with no dimensions or choices. We write $\lfloor e \rfloor_{D.t}$ for the selection of tag t from dimension D in expression e. Tag selection is performed by replacing in e the topmost-leftmost dimension declaration $\textbf{dim}\ D\langle t_1,t_2\rangle\ \textbf{in}\ e'$ with a version of e' that is obtained by substituting choices bound by D with either their first or second alternatives (depending on whether $t = t_1$ or $t = t_2$). If e does not contain a dimension D, it remains unchanged.

A *decision* is a sequence of dimension-qualified tags. A decision that produces a plain expression is called a *complete decision*. The (selection) semantics $[\![e]\!]$ of an expression e is then a mapping from complete decisions to plain lambda calculus expressions.

$$
\begin{aligned}
&[\![\textbf{dim}\ A\langle t_1,t_2\rangle\ \textbf{in}\ A\langle \lambda x.x, \lambda y.\textbf{dim}\ B\langle t_3,t_4\rangle\ \textbf{in}\ B\langle 2,3\rangle\rangle]\!] = \\
&\quad \{([A.t_1],\lambda x.x),([A.t_2,B.t_3],\lambda y.2),([A.t_2,B.t_4],\lambda y.3)\}
\end{aligned}
$$

[2] We omit here for simplicity two constructs for sharing since they do not affect the type system in any way.

Note that tags in dimension A always occur before tags in dimension B in the domain of the mapping. Also, note that dimension B does not appear at all in the first decision since it is eliminated by the selection of the tag $A.t_1$.

3. Partial Variational Types

In Section 1 we motivated the use of *variational types* for typing variational programs. In this section we extend this representation to support *partial variational types*, that is, variational types that contain type errors. The extended representation is given below.

T	::=	τ	*Constant Type*
	|	a	*Type Variable*
	|	$T \to T$	*Function Type*
	|	$D\langle T, T\rangle$	*Choice Type*
	|	$\bot$	*Error Type*
	|	$\top$	*OK Type*

Constant types, type variables, and function types are as in other type systems—*plain types* contain only these three constructs.

Non-plain types may also contain *choice types*. Choice types encode variation in types in the same way that choices encode variation in expressions, with the exception that dimension names in types are globally scoped (see [5] for the rationale). Choice types often correspond directly to choice expressions; for example, the subexpression $A\langle \lambda x.\texttt{true}, \texttt{3}\rangle$ might have the corresponding choice type $A\langle a \to \texttt{Bool}, \texttt{Int}\rangle$. Since there are no tags at the type level, we extend selection to types by writing $\lfloor T \rfloor_{D.i}$, where $i \in \{1,2\}$, to represent selecting the ith alternative in all choices in dimension D. If T contains no such choices, then $\lfloor T \rfloor_{D.i} = T$. We call $D.i$ a *selector* and allow selections on types to be made in any order.

The *error type*, $\bot$, represents a type error and can appear anywhere in a variational type. We say that a variational type is *partial* if it contains one or more error types and *complete* otherwise.

Finally, the symbol $\top$ is used to represent an arbitrary complete type that also contains no type variables, that is, a type that is monomorphic and error-free. This abstraction is only used in *typing patterns*, which are described in the next section.

Many syntactically different types can be considered *equivalent* in that they represent essentially the same mapping from decisions to plain types. Type equivalency is an important concept in typing variational programs. For example, usually when applying a function of type $T \to T'$ to an argument of type T'', we require that $T = T''$, but this requirement is too strict in the variational setting. Consider the expression $\texttt{succ}\, A\langle \texttt{1}, \texttt{2}\rangle$. The type of `succ` is $\texttt{Int} \to \texttt{Int}$ while the type of the argument is $A\langle \texttt{Int}, \texttt{Int}\rangle$. Even though $\texttt{Int} \neq A\langle \texttt{Int}, \texttt{Int}\rangle$, the expression should be considered well-typed because both variants (`succ 1` and `succ 2`) are well-typed. Thus, we say that the two types are *equivalent*, written $\texttt{Int} \equiv A\langle \texttt{Int}, \texttt{Int}\rangle$, and require only equivalency rather equality in well-typed function applications.

Figure 1 gives the type equivalence relation in full. Most of the equivalence rules are straightforward. The FUN and CHOICE rules propagate equivalency across function types and choice types, the F-C rule commutes function types and choice types, and the two SWAP rules commute choice types in different dimensions. The three rules at the bottom of the figure make the relation reflexive, symmetric, and transitive. The two interesting cases are C-IDEMP and the MERGE rules. The C-IDEMP rule captures the property of *choice idempotency*, demonstrated in the example above. The MERGE rules capture the property of *choice domination*. For example, given the choice type $D\langle D\langle T_1, T_2\rangle, T_3\rangle$, we say that the outer choice dominates the inner since there is no way to select type T_2—the selection of the first alternative in the outer choice implies the selection of the first alternative in the inner choice. Note that choice domination only applies to nested choices *in the same dimension*.

$$\textsc{Fun}\ \frac{T_l' \equiv T_r' \qquad T_l \equiv T_r}{T_l' \to T_l \equiv T_r' \to T_r}$$

$$\text{F-C}\ \ D\langle T_1, T_2\rangle \to D\langle T_1', T_2'\rangle \equiv D\langle T_1 \to T_1', T_2 \to T_2'\rangle$$

$$\text{C-C-SWAP1}\ \ D'\langle D\langle T_1, T_2\rangle, T_3\rangle \equiv D\langle D'\langle T_1, T_3\rangle, D'\langle T_2, T_3\rangle\rangle$$

$$\text{C-C-SWAP2}\ \ D'\langle T_1, D\langle T_2, T_3\rangle\rangle \equiv D\langle D'\langle T_1, T_2\rangle, D'\langle T_1, T_3\rangle\rangle$$

$$\text{C-C-MERGE1}\ \ D\langle D\langle T_1, T_2\rangle, T_3\rangle \equiv D\langle T_1, T_3\rangle \qquad \text{C-C-MERGE2}\ \ D\langle T_1, D\langle T_2, T_3\rangle\rangle \equiv D\langle T_1, T_3\rangle$$

$$\textsc{Choice}\ \frac{T_1 \equiv T_1' \qquad T_2 \equiv T_2'}{D\langle T_1, T_2\rangle \equiv D\langle T_1', T_2'\rangle} \qquad \text{C-IDEMP}\ \frac{T_1 \equiv T \qquad T_2 \equiv T}{D\langle T_1, T_2\rangle \equiv T}$$

$$\textsc{Refl}\ \ T \equiv T \qquad \textsc{Symm}\ \frac{T \equiv T'}{T' \equiv T} \qquad \textsc{Trans}\ \frac{T \equiv T' \qquad T' \equiv T''}{T \equiv T''}$$

Figure 1: Variational type equivalence.

In [5] we define a normalization process that can be used to check if two types are equivalent; this can be trivially extended to variational types containing error types. A type is in *normal form* if (1) all function types are maximally distributed into choice types, (2) choice types are nested according to a fixed ordering on dimension names, (3) the alternatives of each choice type are different, and (4) no choice type contains another choice type of the same name. For example, the types $B\langle \texttt{Int}, \texttt{Int}\rangle \to A\langle \texttt{Bool}, \bot\rangle$ and $\texttt{Int} \to A\langle \texttt{Bool}, \bot\rangle$ are not in normal form, but $A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \bot\rangle$ is.

4. Typing Patterns

A typing pattern is a variation type consisting only of $\bot$, $\top$, and choice types and is used to describe *which variants* of an expression are well-typed and which contain type errors. For example, the typing pattern $P = A\langle \top, B\langle \top, \bot\rangle\rangle$ indicates a type error in the variant corresponding to the decision $[A.2, B.2]$, and not in any other variants. A single typing pattern corresponds to an infinite number of partial variational types. Some types corresponding to P include: $A\langle \texttt{Int}, B\langle \texttt{Bool}, \bot\rangle\rangle$, $A\langle \texttt{Int}, \texttt{Bool}\rangle \to B\langle \texttt{Int}, A\langle \texttt{Bool}, \bot\rangle\rangle$, and $A\langle \texttt{Int}, B\langle \texttt{Bool}, \bot\rangle \to B\langle \texttt{Int}, \bot\rangle\rangle$. In these examples, the constant and function types are irrelevant—all that matters is that selecting $[A.2, B.2]$ produces a type containing errors, and that all other type variants are complete.

Typing patterns are not really types in the traditional sense, but rather an *abstraction* of variation types that indicate where the errors are in the variation space. They are useful for determining which types are *more defined* than others (that is, which contain errors in fewer variants) and play a crucial role in the unification of partial types (see Section 7). We conflate the representation of variational types and typing patterns because they behave similarly and doing so allows us to reuse a lot of machinery. In the rest of this section, we employ typing patterns to define a few operations that will be used throughout the paper.

We begin by defining a reflexive, transitive relation for determining which typing patterns are *more defined* than others, given in Figure 2. All typing patterns are more defined than $\bot$ and less defined than $\top$. Note that one typing pattern is not more defined

$$P \leq P \qquad P \leq \bot \qquad \top \leq P \qquad \frac{P \leq P_1 \qquad P \leq P_2}{P \leq D\langle P_1, P_2\rangle}$$

$$\frac{P_1 \leq P \qquad P_2 \leq P}{D\langle P_1, P_2\rangle \leq P} \qquad \frac{P_1 \leq P_1' \qquad P_2 \leq P_2'}{D\langle P_1, P_2\rangle \leq D\langle P_1', P_2'\rangle}$$

Figure 2: The more-defined relation on typing patterns.

than another by simply having fewer occurrences of error types. For example, the pattern $A\langle B\langle \bot, \top\rangle, B\langle \top, \bot\rangle\rangle$ is trivially more defined than $\bot$.

Next, we consider the *masking* of types with patterns. Given a pattern P and a type T, masking $P \lhd T$ potentially adds error types to T according to the position of error types in P.

$$\top \lhd T = T \qquad \bot \lhd T = \bot$$

$$D\langle P_1, P_2\rangle \lhd T = D\langle P_1 \lhd \lfloor T \rfloor_{D.1}, P_2 \lhd \lfloor T \rfloor_{D.2}\rangle$$

For example, masking type $\texttt{Int} \to A\langle \texttt{Bool}, \texttt{Int}\rangle$ with the typing pattern $A\langle \top, \bot\rangle$ yields the type $A\langle \texttt{Int} \to \texttt{Bool}, \bot\rangle$.

The *intersection* of two typing patterns P and P', written $P \otimes P'$, is a pattern that is well-typed in exactly those variants that are well-typed in both P and P'. For example, given patterns $A\langle \top, \bot\rangle$ and $B\langle \bot, \top\rangle$, their intersection is $A\langle B\langle \bot, \top\rangle, \bot\rangle$, which indicates that the only well-typed variant corresponds to the decision $[A.1, B.2]$. Intersection is just a special case of masking, where the masked type is a typing pattern: $P \otimes P' = P \lhd P'$.

The dual of intersection is pattern *union*. The union of two typing patterns P and P', written $P \oplus P'$, is well-typed in those variants that are well-typed in either P or P', or both.

$$\top \oplus P = \top \qquad \bot \oplus P = P$$

$$D\langle P_1, P_2\rangle \oplus P = D\langle P_1 \oplus \lfloor P \rfloor_{D.1}, P_2 \oplus \lfloor P \rfloor_{D.2}\rangle$$

For example, the union of $A\langle \top, \bot\rangle$ and $B\langle \bot, \top\rangle$ is $A\langle \top, B\langle \bot, \top\rangle\rangle$.

Note that the above definitions are all left-biased with regard to the nesting order of choices and the structure of the resulting type. This bias can be eliminated through the normalization process described in [5], which can be applied unaltered to typing patterns.

In the typing process, we often need to check whether two types *match*, for example, to check that the argument type of a function matches the type of the argument it is applied to. Rather than a simple boolean response, we can use typing patterns to provide a more precise account, indicating in which variants the types match ($\top$) and in which they do not ($\bot$). In the following definition of the variational type matching operation $\bowtie : T \times T \to P$ we assume both arguments are in normal form.[3]

$$\begin{aligned}
T \bowtie T &= \top \\
T_1 \to T_1' \bowtie T_2 \to T_2' &= T_1 \bowtie T_2 \otimes T_1' \bowtie T_2' \\
D\langle T_1, T_2\rangle \bowtie D\langle T_1', T_2'\rangle &= D\langle T_1 \bowtie T_1', T_2 \bowtie T_2'\rangle \\
D\langle T_1, T_2\rangle \bowtie T &= D\langle T_1 \bowtie T, T_2 \bowtie T\rangle \\
T \bowtie D\langle T_1, T_2\rangle &= D\langle T_1, T_2\rangle \bowtie T \\
\bot \bowtie T = T \bowtie \bot &= \bot \\
T \bowtie T' &= \bot \qquad \textit{(otherwise)}
\end{aligned}$$

For example, matching $\texttt{Int} \to A\langle \texttt{Bool}, \bot\rangle \bowtie B\langle \texttt{Int}, \bot\rangle \to \texttt{Bool}$ produces the typing pattern $A\langle B\langle \top, \bot\rangle, \bot\rangle$. This operation is used in the typing of applications, as we'll see in the next section.

[3] Assumed for this presentation only. Type matching is actually part of the unification algorithm, whose arguments need not be in normal form.

$$\textsc{T-Con}\ \frac{c \text{ is a constant of type } \tau}{\Delta, \Gamma \vdash c : \tau} \qquad \textsc{T-Abs}\ \frac{\Delta, \Gamma; (x, T') \vdash e : T}{\Delta, \Gamma \vdash \lambda x.e : T' \to T} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = T}{\Delta, \Gamma \vdash x : T}$$

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

$$\textsc{T-Dim}\ \frac{\Delta; (D, D'), \Gamma \vdash e : T \qquad D' \text{ is fresh}}{\Delta, \Gamma \vdash \textbf{dim}\ D\langle t_1, t_2\rangle\ \textbf{in}\ e : T}$$

$$\textsc{T-Choice}\ \frac{\Delta, \Gamma \vdash e_1 : T_1 \qquad \Delta, \Gamma \vdash e_2 : T_2 \qquad \Delta(D) = D'}{\Delta, \Gamma \vdash D\langle e_1, e_2\rangle : D'\langle T_1, T_2\rangle}$$

Figure 3: Typing rules mapping VLC expressions to partial types.

5. An Error-Tolerant Type System

The association of variational types with VLC expressions is determined by a set of typing rules, given in Figure 3. A VLC typing judgment has the form $\Delta, \Gamma \vdash e : T$, which states that expression e has type T in the context of environments Δ and Γ. Environments are implemented as stacks, where $E; (k, v)$ means to push the mapping (k, v) onto environment E, and $E(k) = v$ means that the topmost occurrence of k is mapped to v in E. The Γ environment maps variables to types and is the standard typing environment for lambda calculus. It is used as expected in the typing rules for variables and abstractions. The Δ environment maps expression-level dimension names to globally unique type-level dimension names. These mappings are added by the T-Dim rule and referenced by the T-Choice rule. The use of this environment also ensures that every choice is in scope of a corresponding dimension.

The focus here is on the T-App rule for typing applications, extending it to support partial types. Previously this rule required that the left argument be equivalent to a function type whose argument type is unifiable with the type of the parameter value. In the presence of partial types, we can relax these requirements, introducing error types (rather than failing) when they are not satisfied.

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

$$\begin{aligned}
\uparrow(T_1 \to T_2) &= T_1 \to T_2 \\
\uparrow(D\langle T_1 \to T_1', T_2 \to T_2'\rangle) &= D\langle T_1, T_2\rangle \to D\langle T_1', T_2'\rangle \\
\uparrow(D\langle T_1, T_2\rangle) &= \uparrow(D\langle \uparrow(T_1), \uparrow(T_2)\rangle) \\
\uparrow(T) &= \bot \to \bot \qquad \textit{(otherwise)}
\end{aligned}$$

For example, $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Int}\rangle) = A\langle \texttt{Int}, \texttt{Bool}\rangle \to A\langle \texttt{Bool}, \texttt{Int}\rangle$, while $\uparrow(A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int}\rangle)$ must introduce error types to lift the function type to the top: $A\langle \texttt{Int}, \bot\rangle \to A\langle \texttt{Bool}, \bot\rangle$.

To illustrate the typing of an application, consider the expression $e_1\ e_2$, where $e_1 : A\langle \texttt{Int} \to \texttt{Bool}, \texttt{Bool} \to \texttt{Bool}\rangle$ and $e_2 : \texttt{Int}$. Applying $\uparrow$ to the type of e_1 and simplifying the result type yields the type $A\langle \texttt{Int}, \texttt{Bool}\rangle \to \texttt{Bool}$. Matching $A\langle \texttt{Int}, \texttt{Bool}\rangle \bowtie \texttt{Int}$ pro-

duces the typing pattern $A\langle\top,\bot\rangle$, which we use to mask the result, $A\langle\top,\bot\rangle \lhd \texttt{Bool}$, producing the type of the application: $A\langle\texttt{Bool},\bot\rangle$.

The previous T-App rule emerges as a special case of the generalized one. When e_1 is a function type whose argument type matches the type of e_2, then matching returns $\top$ and masking doesn't alter the return type.

The correspondence between variational types and VLC expressions is established inductively through the process of selection. Given that $e : T$, if e is plain, then T is a plain type or $\bot$. If e is not plain, then we can select a tag from e to produce $e' : T'$, and T' can be obtained by a corresponding selection from T. The inductive step is captured in the following lemma, which can be proved by induction over typing derivations.

Lemma 1 (Variation elimination).
$\Delta,\Gamma \vdash e : T \implies \forall\, D,t : \Delta,\Gamma \vdash \lfloor e \rfloor_{D.t} : \lfloor T \rfloor_{\varphi_e([D.t])}$

Since tags are not present at the type level, and since expression-level dimension names may differ from type-level ones, the function φ_e is a function derived from e that maps tag sequences to the set of corresponding type-level selectors.

By induction it follows that a sequence of selections that produces a plain expression can be used to select a corresponding plain or error type. This results in the following theorem, where $\overline{q}$ is a list of dimension-qualified tags and $\overline{s}$ is a list of type-level selectors.

Theorem 1 (Type preservation). *If* $\varnothing,\Gamma \vdash e : T$ *and* $(\overline{q},e') \in [\![e]\!]$, *then* $\varnothing,\Gamma \vdash e' : T'$ *where* $\varphi_e(\overline{q}) = \overline{s}$ *and* $(\overline{s},T') \in [\![T]\!]$.

This theorem demonstrates the soundness of the type systems since it establishes that from the type of a variational program we can obtain the type of each program variant it contains. We had similar type preservation results in [5], but they applied to only well-typed variational programs. The results here are stronger since they apply to *any* variational programs.

6. The Unification of Partial Types

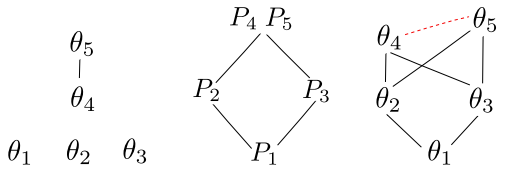
Having extended the type system to work with and produce partial types, we now turn to the more challenging problem of inferring variational types containing type errors. By far the most difficult piece is partial type unification. In Section 6.1 we will describe the specific challenges posed. In particular, the unification algorithm must yield unifiers that produce types that are both most-general *and* most-defined, two qualities that are not obviously orthogonal. In Section 6.2 we show that such unifiers exist, and in Section 7 we present an algorithm for computing unifiers.

6.1 Reconciling Type Partiality and Generality

To support partial type inference, we must extend variational type unification to produce and extend mappings containing error types, and to identify mappings that are somehow best.

As a running example, consider the application $e\ e'$ where $e : T = A\langle\texttt{Int},\texttt{Bool}\rangle \to a$ and $e' : T' = B\langle\texttt{Int},a\rangle$. Usually we would find the most general unifier (mgu) for the problem $A\langle\texttt{Int},\texttt{Bool}\rangle \equiv^? B\langle\texttt{Int},a\rangle$, but in this case the two types are not unifiable since there is a type error in the $[A.2,B.1]$ variant. So what should we map a to? The mapping we choose should be *most-general* in the usual sense, but it should also be *most-defined*, yielding types with type errors in as few variants as possible. In this subsection we will explore the interaction of these two properties.

In Figure 4 we list several mappings we might choose to partially unify T and T' in our example. In the table, the type constants `Bool`, `Char`, and `Int` are shortened for space reasons. Each mapping is identified by a θ_i, for example, $\theta_2 = \{a \mapsto \texttt{Int}\}$. We also give the result of applying each mapping to each of the two types as T_i and T_i', the typing pattern P_i that results from matching the argument



a. θ_i by $\sqsubseteq$ b. P_i by $\leq$ c. θ_i by $\leq, \sqsubseteq$

Figure 5: Orderings among patterns, result types, and mappings.

type of T_i to T_i', and the result type generated by masking the result type of T_i with P_i. Note that we apply mappings by adjacency and use the functions *arg* and *res* to access, respectively, the argument and result types of a function type.

Figure 5 visualizes the more-general and more-defined relationships among mappings and typing patterns. The relations are defined for elements connected by lines, and the element higher in the graph is considered more general or more defined.

The first thing to note is that the standard more-general relation, $\sqsubseteq$, is not very helpful in selecting a mapping. A mapping θ is more general than θ', written $\theta \sqsubseteq \theta'$ if $\exists\theta''$ such that $\theta' = \theta'' \circ \theta$. But this relationship is only defined on one pair of our five mappings: $\theta_5 \sqsubseteq \theta_4$ (since $\{b \mapsto A\langle\texttt{Int},\texttt{Bool}\rangle\} \circ \theta_5 = \theta_4$). Since we are not restricted to mappings that are valid unifiers, there are many more possibilities, and many will not be ordered by the more-general relation.

More useful is the more-defined relation (see Section 4) on the match-produced typing patterns, for which many relationships are defined, as seen in Figure 5b. Using this metric, we can rule out mappings θ_1, θ_2, and θ_3 because they will produce types with errors in more variants than the mappings θ_4 and θ_5. The problem is that θ_4 and θ_5 produce the same pattern.

The solution, of course, is to use both metrics together, as demonstrated in Figure 5c. The solid lines between mappings correspond to more-defined relations between the generated typing patterns, and the dotted line corresponds to the more-general relation between the mappings directly. This reveals θ_5 as the most-defined, most-general mapping.

At this point it is not clear whether this convergence was a quirk of our example, or whether these properties will always converge in this way. In the next section we will tackle the general case, and show that a most-defined, most-general mapping always exists.

6.2 Most-General Partial Unifiers

In Section 6.1, we have illustrated how unification with partial types requires the integration of two partial orderings of types, $\leq$ and $\sqsubseteq$. In this section, we introduce the necessary machinery that enables unification to deal with this situation in general and produce most general partial unifiers.

In the following we consider a general unification problem of the form $U = T_L \equiv^? T_R$. For a given mapping θ, we write $U :: \theta$ for the typing pattern $T_L\theta \bowtie T_R\theta$ that results from θ and U. When we say that P is a typing pattern for U, we mean that there is some θ such that $P = U :: \theta$. With $vars(U)$ we refer to all type variables in U, and we use $dom(\theta)$ to denote the domain of θ. We use $\|\theta\|_U$ to normalize θ with respect to the variables in U, that is, $\|\theta\|_U$ is obtained from θ by renaming type variables such that $dom(\|\theta\|_U) = vars(U)$.

Finally, we extend selection to apply to unification problems and mappings, that is, $\lfloor U \rfloor_{D.i} = \lfloor T_L \rfloor_{D.i} \equiv^? \lfloor T_R \rfloor_{D.i}$ and $\lfloor\theta\rfloor_{D.i} = \{(a, \lfloor T \rfloor_{D.i}) \mid (a,T) \in \theta\}$. We write $\theta|_V$ for the restriction of θ by a set of variables V, which is defined as $\theta|_V = \{(a,a\theta) \mid a \in V\}$.

The first three lemmas state that selection extends in a homomorphic way across several operations.

i	θ_i	$T_i = T\theta_i$	$T_i' = T'\theta_i$	$P_i = arg(T_i) \bowtie T_i'$	$R_i = P_i \lhd res(T_i)$
1	$\{a \mapsto \texttt{Ch}\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to \texttt{Ch}$	$B\langle \texttt{In}, \texttt{Ch}\rangle$	$A\langle B\langle \top, \bot\rangle, \bot\rangle$	$A\langle B\langle \texttt{Ch}, \bot\rangle, \bot\rangle$
2	$\{a \mapsto \texttt{In}\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to \texttt{In}$	$B\langle \texttt{In}, \texttt{In}\rangle$	$A\langle \top, \bot\rangle$	$A\langle \texttt{In}, \bot\rangle$
3	$\{a \mapsto \texttt{Bo}\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to \texttt{Bo}$	$B\langle \texttt{In}, \texttt{Bo}\rangle$	$A\langle B\langle \top, \bot\rangle, B\langle \bot, \top\rangle\rangle$	$A\langle B\langle \texttt{Bo}, \bot\rangle, B\langle \bot, \texttt{Bo}\rangle\rangle$
4	$\{a \mapsto A\langle \texttt{In}, \texttt{Bo}\rangle\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to A\langle \texttt{In}, \texttt{Bo}\rangle$	$B\langle \texttt{In}, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle$	$A\langle \top, B\langle \bot, \top\rangle\rangle$	$A\langle \texttt{In}, B\langle \bot, \texttt{Bo}\rangle\rangle$
5	$\{a \mapsto B\langle b, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle\}$	$A\langle \texttt{In}, \texttt{Bo}\rangle \to B\langle b, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle$	$B\langle \texttt{In}, A\langle \texttt{In}, \texttt{Bo}\rangle\rangle$	$A\langle \top, B\langle \bot, \top\rangle\rangle$	$A\langle B\langle b, \texttt{In}\rangle, B\langle \bot, \texttt{Bo}\rangle\rangle$

Figure 4: Some mappings for $T = A\langle \texttt{Int}, \texttt{Bool}\rangle \to a$ and $T' = B\langle \texttt{Int}, a\rangle$, with the typing pattern and result types they produce.

LEMMA 2. $\lfloor T_L \bowtie T_R \rfloor_{D.i} = \lfloor T_L \rfloor_{D.i} \bowtie \lfloor T_R \rfloor_{D.i}$

The proofs for this and the following lemmas, left out for brevity, proceed by applying the definition of the operation under consideration and then performing structural induction on types.

LEMMA 3.
$$\lfloor T_L \oplus T_R \rfloor_{D.i} = \lfloor T_L \rfloor_{D.i} \oplus \lfloor T_R \rfloor_{D.i}$$
$$\lfloor T_L \otimes T_R \rfloor_{D.i} = \lfloor T_L \rfloor_{D.i} \otimes \lfloor T_R \rfloor_{D.i}$$
$$\lfloor P \lhd T \rfloor_{D.i} = \lfloor P \rfloor_{D.i} \lhd \lfloor T \rfloor_{D.i}$$
$$\lfloor T_L \to T_R \rfloor_{D.i} = \lfloor T_L \rfloor_{D.i} \to \lfloor T_R \rfloor_{D.i}$$

We also have a similar result for type substitution.

LEMMA 4. $\lfloor T\theta \rfloor_{D.i} = \lfloor T \rfloor_{D.i} \lfloor \theta \rfloor_{D.i}$

The next lemma says that the computation of typing patterns can be decomposed by using selection.

LEMMA 5. $\lfloor U :: \theta \rfloor_{D.i} = \lfloor U :: \lfloor \theta \rfloor_{D.i} \rfloor_{D.i} = \lfloor U \rfloor_{D.i} :: \lfloor \theta \rfloor_{D.i}$

PROOF. The proof for the first part is as follows. Let $P = U :: \theta$ and $P' = U :: \lfloor \theta \rfloor_{D.i}$, then

$$\begin{aligned}
\lfloor P \rfloor_{D.i} &= \lfloor T_L\theta \bowtie T_R\theta \rfloor_{D.i} \\
&= \lfloor T_L\theta \rfloor_{D.i} \bowtie \lfloor T_R\theta \rfloor_{D.i} && \text{by Lemma 2} \\
&= \lfloor T_L \rfloor_{D.i} \lfloor \theta \rfloor_{D.i} \bowtie \lfloor T_R \rfloor_{D.i} \lfloor \theta \rfloor_{D.i} && \text{by Lemma 4} \\
\lfloor P' \rfloor_{D.i} &= \lfloor T_L \lfloor \theta \rfloor_{D.i} \bowtie T_R \lfloor \theta \rfloor_{D.i} \rfloor_{D.i} \\
&= \lfloor T_L \lfloor \theta \rfloor_{D.i} \rfloor_{D.i} \bowtie \lfloor T_R \lfloor \theta \rfloor_{D.i} \rfloor_{D.i} && \text{by Lemma 2} \\
&= \lfloor T_L \rfloor_{D.i} \lfloor \lfloor \theta \rfloor_{D.i} \rfloor_{D.i} \bowtie \lfloor T_R \rfloor_{D.i} \lfloor \lfloor \theta \rfloor_{D.i} \rfloor_{D.i} && \text{by Lemma 4} \\
&= \lfloor T_L \rfloor_{D.i} \lfloor \theta \rfloor_{D.i} \bowtie \lfloor T_R \rfloor_{D.i} \lfloor \theta \rfloor_{D.i}
\end{aligned}$$

The proof for the second part is analogous. □

LEMMA 6 (Typing patterns have a join). *If P_1 and P_2 are typing patterns for U, then so is $P_1 \oplus P_2$.*

PROOF. Assume θ_1 and θ_2 are the mappings such that $P_1 = U :: \theta_1$ and $P_2 = U :: \theta_2$. The proof consists of several cases. For each case, we construct a mapping θ_3 such that $U :: \theta_3 = P_1 \oplus P_2$, which we denote as P_3. We show the proof for the case where $P_1 = D\langle P_{11}, P_{12}\rangle$ and $P_2 = D\langle P_{21}, P_{22}\rangle$ and there is no $\trianglelefteq$ relation between P_1 and P_2. The proofs for other cases are simpler or can be transformed into this case. We assume that θ_1 and θ_2 are already normalized with respect to U. We can consider several cases.

First, if we assume $P_{21} \trianglelefteq P_{11}$ and $P_{12} \trianglelefteq P_{22}$, we let $\theta_3 = \{(a, D\langle \lfloor a\theta_2 \rfloor_{D.1}, \lfloor a\theta_1 \rfloor_{D.2}\rangle) \mid a \in vars(U)\}$, for which we observe the following.

$$\begin{aligned}
U :: \theta_3 &= D\langle \lfloor U :: \theta_3 \rfloor_{D.1}, \lfloor U :: \theta_3 \rfloor_{D.2}\rangle \\
&= D\langle \lfloor U \rfloor_{D.1} :: \lfloor \theta_3 \rfloor_{D.1}, \lfloor U \rfloor_{D.2} :: \lfloor \theta_3 \rfloor_{D.2}\rangle && \text{Lemma 5} \\
&= D\langle \lfloor U \rfloor_{D.1} :: \lfloor \theta_1 \rfloor_{D.1}, \lfloor U \rfloor_{D.2} :: \lfloor \theta_2 \rfloor_{D.2}\rangle && \text{construction} \\
&= D\langle \lfloor U :: \theta_1 \rfloor_{D.1}, \lfloor U :: \theta_2 \rfloor_{D.2}\rangle && \text{Lemma 5} \\
&= D\langle P_{21}, P_{12}\rangle \\
&= P_1 \oplus P_2 && \text{def. of } \oplus
\end{aligned}$$

Second, the case for $P_{11} \trianglelefteq P_{21}$ and $P_{22} \trianglelefteq P_{12}$ is analogous.

Third, if there is no $\trianglelefteq$ relation between P_{21} and P_{11} or P_{12} and P_{22}, we let $U_1 = \lfloor U \rfloor_{D.1}$, $U_2 = \lfloor U \rfloor_{D.2}$, $\theta_{11} = \theta_1|_{vars(U_1)}$, $\theta_{12} = \theta_1|_{vars(U_2)}$, $\theta_{21} = \theta_2|_{vars(U_1)}$ and $\theta_{22} = \theta_2|_{vars(U_2)}$. By induction, we can construct a mapping θ_{31} from θ_{11} and θ_{21} for U_1 such that $U_1 :: \theta_{31} = P_{11} \oplus P_{21}$. Likewise, we can construct a mapping θ_{32} from θ_{12} and θ_{22} for U_2 such that $U_2 :: \theta_{32} = P_{12} \oplus P_{22}$. We can now build θ_3 based on θ_{31} and θ_{32} as follows. For each type variable $a \in vars(U)$ we define θ_3 as follows.

$$\theta_3(a) = \begin{cases} D\langle a\theta_{31}, a\theta_{32}\rangle & \text{if } a \in vars(U_1) \wedge a \in vars(U_2) \\ a\theta_{31} & \text{if } a \in vars(U_1) \\ a\theta_{32} & \text{if } a \in vars(U_2) \end{cases}$$

Proving that $U :: \theta_3 = D\langle P_{31}, P_{32}\rangle = D\langle P_{11}, P_{12}\rangle \oplus D\langle P_{21}, P_{22}\rangle$ is similar to the proof for the previous case. □

Combining this lemma with the rule $\top \trianglelefteq P$ we can conclude that for any unification problem, there is an upper-bound typing pattern, which we call the *principal typing pattern*.

THEOREM 2 (Existence of principal typing patterns). *For every unification problem U there is a mapping θ with $P = U :: \theta$, such that $P \trianglelefteq P'$ for any other mapping θ' with $P' = U :: \theta'$.*

We call a mapping that leads to the principal typing pattern a *partial unifier* and use η to denote partial unifiers. We call mappings that are not partial unifiers "non-unifiers" for short. Based on these definitions, the first example in Section 6.1 has the principal typing pattern P_4 and partial unifiers θ_4 and θ_5.

Theorem 2 only shows the existence of partial unifiers, but does not say anything about how many partial unifiers exist and how they are possibly related. It turns out that partial unifiers can be compared with respect to their generality and for each unification problem there is a *most general partial unifier* (mgpu) of which all other partial unifiers are instances.

THEOREM 3 (Partial unification is unitary). *For every unification problem U there is one partial unifier η of such that any other partial unifier η' for U is an instance of it, that is, $\eta \sqsubseteq \eta'$.*

The proof strategy is similar to that for Theorem 2, although more complex. Given any two partial unifiers, we can construct a new partial unifier that is more general than the old ones.

7. A Unification Algorithm

In this section we present a partial type unification algorithm that identifies partial unifiers that produce most-general, most-defined types. This algorithm is a conservative extension of our algorithm for unifying complete variational types, presented in [5]. That is, when the types are complete and fully unifiable, we produce the same results as before. When the types to be unified are partial and/or not unifiable, we produce partial unifiers as described in the previous section. In Section 7.1 we give a high-level overview of the process of unifying variational types, and in Section 7.2 we define the algorithm that makes up the core of this process.

7.1 Unification of Variational Types

The fundamental difference between traditional type unification [3] and variational unification is the treatment of type variables. Consider the unification problem $A\langle \texttt{Int}, a\rangle \equiv^? A\langle a, \texttt{Bool}\rangle$. At first it may seem that these types are not unifiable—blithe decomposition by alternatives yields the subproblems $\texttt{Int} \equiv^? a$ and $a \equiv^? \texttt{Bool}$, but a cannot map to both `Int` and `Bool`. However, there *is* a unifier to the original problem: if we map a to $A\langle \texttt{Int}, \texttt{Bool}\rangle$, then both types are equivalent to $A\langle \texttt{Int}, \texttt{Bool}\rangle$ by choice domination (see Section 3). Decomposition is essential to the unification process, but decomposing by alternatives discards important context provided by the choice type. In our example, this context tells us that only one of the two a type variables will be selected in any particular variant.

As a solution, we encode the contextual information in the type variables themselves. A *qualified type variable* is a type variable marked by the choice type alternatives in which it is nested. We write $a_{A\tilde{B}}$ to indicate that type variable a is located in the first alternative of a choice type in dimension A and the second alternative of a choice type in B. Throughout most of the unification process, type variables with different qualifications are simply considered to be different type variables, but we can use the contextual information to construct the final mappings (from unqualified type variables to variational types) through a process called *completion*. In the example above, after qualification and decomposition we identify the mappings $\{a_A \mapsto \texttt{Int}, a_{\tilde{A}} \mapsto \texttt{Bool}\}$ which completes to the final result $\{a \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}$.

Unification thus consists of three main phases: (1) the unification problem U is translated into a corresponding *qualified unification* problem Q, (2) Q is solved, and (3) the solution to Q is completed to produce a solution to U. The first step of this process is trivial. We simply traverse both types and qualify all of the type variables. Completion is also straightforward: given a list of mappings from qualified type variables, each $a_{q_i} \mapsto T_i$ describes a leaf in a tree of nested choice types that makes up the type T in the completed mapping $a \mapsto T$. We just iterate over the qualified mappings, lazily constructing and populating the resulting tree.

The difficult part is of course solving the qualified unification problem. In addition to the traditional operations of matching and decomposition, qualified unification relies on two additional operations. First, a choice type can be *hoisted* over another choice type. For example, hoisting transforms $A\langle T_1, B\langle T_2, T_3\rangle\rangle$ into $B\langle A\langle T_1, T_2\rangle, A\langle T_1, T_3\rangle\rangle$. Second, a type variable can be *split* into a choice type between two qualified versions of that variable. For example, splitting transforms a into $A\langle a_A, a_{\tilde{A}}\rangle$). These operations manipulate the types being unified so they can be further matched or decomposed. For example, the problem $A\langle \texttt{Int}, a_{\tilde{A}}\rangle \equiv^? B\langle b_B, c_{\tilde{B}}\rangle$ cannot be directly decomposed. However, if we split the variable $a_{\tilde{A}}$ into $B\langle a_{\tilde{A}B}, a_{\tilde{A}\tilde{B}}\rangle$ and hoist this choice type to the top, we get the new problem $B\langle A\langle \texttt{Int}, a_{\tilde{A}B}\rangle, A\langle \texttt{Int}, a_{\tilde{A}\tilde{B}}\rangle\rangle \equiv^? B\langle b_B, c_{\tilde{B}}\rangle$, which can be decomposed into two trivial subproblems.

The full technical exposition of the unification of complete variational types is provided in [5]. Significantly, we also show that the unification problem is decidable and unitary. In the rest of this section we will develop the unification of *partial* variational types.

7.2 Computing the Most General Partial Unifier

In Section 6.2 we showed that for each partial unification problem, there is a unique mgpu that produces the corresponding principal typing pattern. In this section, we show how to compute each of these by extending the process described in Section 7.1. We do this first by example, then give the algorithm directly.

Consider the unification problem $A\langle \texttt{Int}, a\rangle \equiv^? B\langle \texttt{Bool}, b\rangle$. We begin, as described in Section 7.1, by transforming this into the corresponding qualified unification problem shown at the top of

$$A\langle \texttt{Int}, a_{\tilde{A}}\rangle \equiv^? B\langle \texttt{Bool}, b_{\tilde{B}}\rangle$$
$$\downarrow_{split}$$
$$A\langle \texttt{Int}, a_{\tilde{A}}\rangle \equiv^? B\langle \texttt{Bool}, A\langle b_{A\tilde{B}}, b_{\tilde{A}\tilde{B}}\rangle\rangle$$
$$\downarrow_{hoist}$$
$$A\langle \texttt{Int}, a_{\tilde{A}}\rangle \equiv^? A\langle B\langle \texttt{Bool}, b_{A\tilde{B}}\rangle, B\langle \texttt{Bool}, b_{\tilde{A}\tilde{B}}\rangle\rangle$$

$$\frac{\texttt{Int} \equiv^? B\langle \texttt{Bool}, b_{A\tilde{B}}\rangle \qquad a_{\tilde{A}} \equiv^? B\langle \texttt{Bool}, b_{\tilde{A}\tilde{B}}\rangle}{*\,\texttt{Int} \equiv^? \texttt{Bool}\,* \quad \texttt{Int} \equiv^? b_{A\tilde{B}}}$$

Figure 6: Qualified unification resulting in a type error.

Figure 6. Since the top-level choice names don't match, we choose a type variable and apply the split-hoist strategy (first two steps) in order to decompose by alternatives (third step). This gives us the two subproblems at the fourth level from the top. When a plain type is unified with a choice type, we can decompose it by unifying the plain type with each alternative. This is demonstrated in the left branch, which yields two smaller subproblems, one of which, $\texttt{Int} \equiv^? \texttt{Bool}$, reveals a type error.

This decomposition contains all of the information needed to construct both the mgpu and the principal typing pattern. We construct the mgpu by composing the mappings generated at the end of every successful branch of the unification process. In this case, there were two successful branches, giving the following mgpu.

$$\{a_{\tilde{A}} \mapsto B\langle \texttt{Bool}, b_{\tilde{A}\tilde{B}}\rangle, b_{A\tilde{B}} \mapsto \texttt{Int}\}$$

We construct the principal typing pattern by observing which branches of the decomposition fail and succeed. In this case, the branch corresponding to the first alternative in both A and B failed, yielding the principal error pattern $A\langle B\langle \bot, \top\rangle, \top\rangle$.

As the final step, we use completion to produce the solution to the original (unqualified) unification problem.

$$\{a \mapsto A\langle c, B\langle \texttt{Bool}, d\rangle\rangle, b \mapsto B\langle f, A\langle \texttt{Int}, d\rangle\rangle\}$$

Figure 7 gives the partial unification algorithm. It accepts a qualified unification problem $T_L \equiv^? T_R$ and returns a principal typing pattern P and a mgpu η. We show only the cases that differ significantly from the qualified unification algorithm presented in [5].

The algorithm relies on several helper functions. The function $choices(T)$ returns the dimension names of all choice types that occur in T. The function *splittable* returns the set of 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 (no function types). The function $vars(T)$ returns the set of qualified variables in a type. Finally, the function $sdims(v_q, T)$ returns the set of dimension names not present in q but present in the qualifications of type variables that are more *specific* than v_q. We say that u_p is more specific than v_q if $u = v$ and p can be written as qp' for some nonempty p'. For example, $sdims(a_A, a_{A\tilde{B}} \to \texttt{Int}) = \{B\}$.

We will work through the cases of the *unify* algorithm, from top to bottom. In the body of the algorithm and in these descriptions, T_L and T_R are used to refer to the first and second arguments to *unify*, respectively. We first consider a couple of base cases. Attempting to unify any type and an error type yields an empty mapping and the fully undefined typing pattern $\bot$. This defines the propagation of errors. When unifying two plain types, we defer to the traditional *robinson* unification algorithm [21]. If it succeeds, we return the unifier and the fully defined typing pattern $\top$. If it fails, we return the empty mapping and $\bot$.

When unifying a *ground plain type* g (a type that does not contain choice types or type variables) with a choice type, we just unify g with both alternatives. This is seen in the second decomposition

$unify : T \times T \to P \times \eta$

$unify(\bot, T) = (\bot, \varnothing)$

$unify(p, p')$
$\quad \mid robinson(p, p') = \bot = (\bot, \varnothing)$
$\quad \mid$ otherwise $= (\top, robinson(p, p'))$

$unify(g, D\langle T_1, T_2\rangle) = unify(D\langle g, g\rangle, D\langle T_1, T_2\rangle)$

$unify(D\langle T_1, T_2\rangle, D\langle T_1', T_2'\rangle) =$
$\quad (P_1, \eta_1) \leftarrow unify(T_1, T_1')$
$\quad (P_2, \eta_2) \leftarrow unify(T_2, T_2')$
$\quad$ return $(D\langle P_1, P_2\rangle, \eta_1 \circ \eta_2)$

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

$unify(v_q, T_1' \to T_2')$
$\quad \mid v_q \in vars(T_R) = (\bot, \varnothing)$
$\quad \mid D \in sdims(v_q, T_R) = unify(D\langle v_{Dq}, v_{\bar{D}q}\rangle, T_R)$
$\quad \mid$ otherwise $= (\top, \{v_q \mapsto T_R\})$

$unify(T_1 \to T_2, T_1' \to T_2') =$
$\quad (P_1, \eta_1) \leftarrow unify(T_1, T_1')$
$\quad (P_2, \eta_2) \leftarrow unify(T_2\eta_1, T_2'\eta_1)$
$\quad P \leftarrow P_1 \otimes P_2$
$\quad$ return $(P, \eta_1 \circ \eta_2)$

Figure 7: Partial unification algorithm.

in Figure 6. The first decomposition is by alternatives, which is performed when unifying two choices in the same dimension; this is captured in the fourth case of *unify*. Note that we do not need to apply the mapping η_1 to T_2 and T_2', as we might expect, because $(vars(T_1) \cup vars(T_1')) \cap (vars(T_2) \cup vars(T_2')) = \varnothing$ due to type variable qualification. We then compose the corresponding unifiers and combine the error patterns with a choice type.

The fifth case considers the unification of two choice types in different dimensions with no splittable type variables. This is not fully unifiable and so would usually represent failure. However, with partial unification we can proceed by attempting to unify all combinations of alternatives in order to locate the variants that contain errors. For example, $A\langle \texttt{Int}, \texttt{Bool}\rangle \equiv^? B\langle \texttt{Int}, \texttt{Bool}\rangle$ produces the typing pattern $A\langle B\langle \top, \bot\rangle, B\langle \bot, \top\rangle\rangle$. We reuse our existing machinery by duplicating T_R and putting it in a choice type that will be decomposed by alternatives in the recursive execution of *unify*.

Although we do not show all of the cases of unifying a qualified type variable against other types, we do show the trickiest case of unifying a type variable with a function type in the sixth case in Figure 7. There are there sub-cases to consider: (1) If v_q occurs in T_R, the unification fails. (2) If v_q does not occur in T_R but a more specific type variable v_{qr} does, then some variants may still be well-typed. So, we create a new unification problem by adding a dimension D from r to the qualification of v_q, then splitting the new variable v_{Dq} in the D dimension. (3) Finally, if v does not appear in any form in T_R, then we simply map v_q to T_R. Note that the decomposition of the unification problem is such that if there is any v_p in T_R, then either $v_p = v_q$ or v_p is more specific than v_q.

Finally, we consider the unification of two function types. We unify the corresponding argument types and result types and compose the mappings. The resulting typing pattern is the intersection of the patterns of the two subproblems since the result will be well-typed only if both the argument and result types agree.

We conclude by presenting some important properties of the unification algorithm. The first result is that the partial unification algorithm is terminating through decomposition that eventually results in either the propagation of type errors, or calls to the *robinson* algorithm, which is terminating. There are two cases that do not decompose, but rather grow the size of the types being unified, and so pose a threat to termination. The first is the splitting of type variables. The second is the fifth case shown in Figure 7. Both of these cases introduce a new choice type and duplicate one of their arguments. These cases do not prevent termination, however, for two reasons. First, both cases are followed immediately by a decomposition that produces two subproblems smaller than the original problem. Second, the number of new choice types that can be introduced is bounded by the overall number of dimensions in the unification problem. This follows from the property of choice domination and the fact that we eliminate a dimension from consideration with each decomposition by alternatives.

In [5], we did an in-depth time complexity analysis of the variational unification algorithm. We showed that if the size of T_L and T_R are l and r respectively, then the time complexity of variational unification is $O(lr(l+r))$. Since the computation of typing patterns in the unification algorithm does not exceed the time for computing partial unifiers, the run-time complexity is still $O(lr(l+r))$ for the partial unification algorithm.

The partial unification algorithm is also sound and complete. These facts are expressed in the following theorems. We use *unify'* to refer to the entire three-part unification process described in Section 7.1 (qualification, qualified unification, completion).

THEOREM 4 (Partial unification is sound). *Given the unification problem $T_1 \equiv^? T_2$, if $unify'(T_1, T_2) = (P, \eta)$, then $T_1\eta \bowtie T_2\eta = P$.*

THEOREM 5 (Partial unification is complete, most defined, and most general). *Given the unification problem $T_1 \equiv^? T_2$, if $T_1\theta \bowtie T_2\theta = P$, then $unify'(T_1, T_2) = (P', \eta)$ such that $P' \trianglelefteq P$ and if $P' \equiv P$ then there exists some θ' such that $\theta = \theta' \circ \eta$.*

8. Partial Type Inference Algorithm

Although the partial unification algorithm is quite complicated, the inference algorithm itself is simple. We define it as an extension of algorithm $\mathcal{W}$ [6] and show the most interesting case below.

$infer : \Delta \times \Gamma \times e \to \eta \times T$
$infer(\Delta, \Gamma, e_1\ e_2) =$
$\quad (\eta_1, T_1) \leftarrow infer(\Delta, \Gamma, e_1)$
$\quad (\eta_2, T_2) \leftarrow infer(\Delta, \Gamma\eta_1, e_2)$
$\quad (P, \eta) \leftarrow unify'(T_1\eta_2, T_2\eta_2 \to a)$ $\quad$ {- *a is a fresh variable* -}
$\quad R \leftarrow P \lhd a\eta$
$\quad$ return $(\eta \circ \eta_2 \circ \eta_1, R)$

The algorithm takes three arguments: a dimension environment, a typing environment, and an expression. It returns a partial unifier and the inferred partial type. Traditionally, inferring the result of a function application consists of four steps: (1) infer the type of the function, (2) infer the type of the argument, (3) unify the argument type of the function with the type of the argument, and (4) instantiate the result type of the function with the returned unifier. Our algorithm adds just one more step: we must mask the result type according to the typing pattern returned by partial unification, in order to introduce error types for the cases where traditional unification would fail.

The remaining cases can be derived from the typing rules in Section 5. Variables and abstractions are treated as in $\mathcal{W}$. For a dimension declaration we extend the dimension environment and recursively infer the type of its scope. For a choice we infer the type of each alternative and build a corresponding choice type.

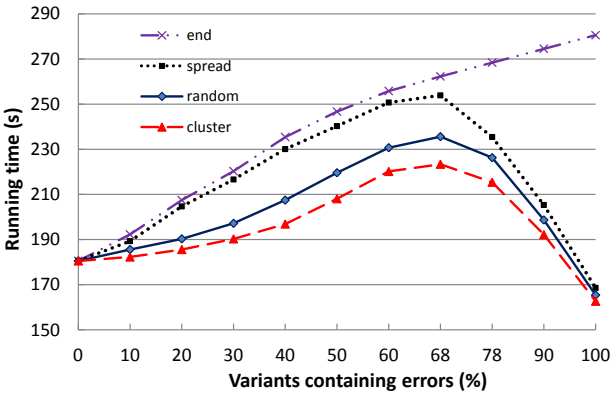


Figure 8: Running time of prototype by error distribution.

The following theorems state that our type inference algorithm is sound and complete and has the principal typing property. In the following, the symbol $\preceq$ represents a more-defined, more-general relation on variational types. That is, $T' \preceq T$ means that for every corresponding pair of (plain) variants V' and V from T' and T, respectively, either $V' \sqsubseteq V$ or $V = \bot$.

THEOREM 6 (Type inference is sound).
If $\mathit{infer}(\Delta,\Gamma,e) = (\eta,T)$, *then* $\Delta,\Gamma\eta \vdash e : T$.

THEOREM 7 (Type inference is complete and principal).
If $\Delta,\Gamma\eta \vdash e : T$, *then* $\mathit{infer}(\Delta,\Gamma,e) = (\eta',T')$ *such that* $\eta = \theta \circ \eta'$ *for some* θ *and* $T' \preceq T$.

These results mean that, for any syntactically correct VLC expression, we can infer the most general type, containing type errors in as few variants as possible, all without type annotations.

9. Evaluation

Variational type inference offers potentially huge efficiency gains over the brute-force strategy of typing each variant individually. The first opportunity is by *sharing* the typing information of code common to multiple variants. For example, in the expression $f\ A\langle e_1,e_2\rangle$ we need only type the function f once relative to e_1 and e_2. The second, more subtle opportunity is by *reducing* variability in types through choice idempotency. This is often possible because types are an abstraction of expressions. For example, in the expression $A\langle f,g\rangle\ B\langle 1,2\rangle$, the argument type reduces to the plain type `Int`, reducing the variability in the application at the type level. While both of these cases are expected to be ubiquitous in practice, there are worst-case scenarios that fundamentally cannot be typed faster than brute-force, for example, an expression with no sharing and in which every variant has a different type, such as $A\langle B\langle T_1,T_2\rangle,B\langle T_3,T_4\rangle\rangle$ where $T_1 \ldots T_4$ are all different.

In this section we empirically evaluate the efficiency of partial type inference in a variety of ways. To do this, we have developed a prototype in Haskell that implements the contents of this paper. The prototype consists of three parts: a normalizer for variational types, the equational partial unification algorithm described in Section 7, and the type inference algorithm described in Section 8.

In the first experiment, we measure the additional cost of the extensions described in this paper, relative to [5]. To measure this overhead effectively, we intentionally induced our worst-case performance through the *cascading choice* problem. Cascading choices are long sequences of applications, where each expression is a choice in a different dimension. If the types of all alternatives are different, no solution can perform better than the brute-force strategy. The overhead of our prototype on such examples (all well-typed, with between 14 and 21 dimensions) was about 30% of the running time of the non-error-tolerant prototype described in [5].

In the second experiment, we study how the distribution of errors in an expression affects the efficiency of partial type inference. The graph in Figure 8 shows the running time of the prototype on a cascading choice problem with 21 dimensions, seeded with errors. The horizontal axis indicates the percentage of variants that were seeded with type errors, and the different lines represent different distributions of these errors. Errors can be *spread* evenly throughout the expression, *clustered* together, distributed *randomly*, or introduced at the *end* of the expression. An interesting phenomenon is that, while the running time at first increases as we introduce errors (due to the costs of maintaining and applying error patterns), in three of the four curves the running time decreases sharply as the error density increases. This is because additional errors introduce opportunities for reduction through choice idempotency ($D\langle\bot,\bot\rangle = \bot$) that are usually denied in cascading choice expressions. As expected, this feature is most pronounced when errors are clustered and least pronounced when they are spread evenly. When errors are introduced at the end of the expression, this opportunity never arises since all the work has already been done.

Finally, in the third experiment, we demonstrate the efficiency and effectiveness of partial type inference in finding type errors, relative to the brute-force approach (implemented as a prototype in the same way as our own). The results are presented in the table in Figure 9. Each row represents an artificially constructed expression that varies in the indicated number of dimensions. The size of each expression is given by the number of AST nodes. The expressions are constructed such that not all dimensions are independent (some dimensions are nested within choices), so the number of variants each expression represents is also given.

In each expression, we manually seeded the indicated number of errors according to two different distributions: errors may be *spread* evenly throughout the expression or *clustered* together. Thus, each row actually represents two expressions with different error distributions that are otherwise identical. Errors are counted relative to the variational expression, not the variants they occur in. For example, if the expression *err* produces a type error, then $A\langle \mathit{err},B\langle 1,2\rangle\rangle$ is considered to contain just one type error even though that error is expressed in two variants ($[A.2,B.1]$ and $[A.2,B.2]$).

Finally, for each expression we give the percentage of errors caught and total running time in seconds (run on a 2.8GHz dual core processor with 3GB RAM) of the brute-force approach and our inference algorithm, respectively. Often the problem is intractable for the brute-force approach, so we cap the running-time at one hour and count the number of errors caught to this point. Because of this cap, and especially when errors are clustered, there is a potential for bias in which variants the brute-force algorithm sees before the time limit is reached. To mitigate this, we ran each brute-force test 10 times, starting from random variants, and averaged the results. Note that for presentation reasons we do not list the percentage of errors found for our algorithm since this value is always 100%. Similarly, we do not list the running time of the brute-force approach since this is the full 3600 seconds in all but a few cases, which are indicated by footnotes.

From the results in Figure 9 we observe that our algorithm scales well as the size, variability, and number of errors in an expression increases. Our algorithm is also more reliable for detecting errors since the ability to completely type the expressions means that it is not sensitive (in this regard) to the distribution of errors, and we do not have to consider issues like which variant the algorithm starts with.

Collectively, these results demonstrate the feasibility of error-tolerant type inference on large, complex expressions. In practice,

size	dims	variants	errors	spread		clustered	
				brute (%)	vlc (s)	brute (%)	vlc (s)
702	22	2^{16}	100	100[a]	0.62	100[c]	0.57
3719	22	2^{16}	100	14.90	1.09	4.10	1.02
976	24	2^{17}	200	100[b]	0.71	100[d]	0.68
5327	24	2^{17}	200	0.50	2.26	39.85	2.17
8412	24	2^{17}	200	0.05	3.79	0.00	3.65
1163	27	2^{21}	400	27.05	0.76	4.90	0.71
1745	33	2^{25}	500	0.42	1.31	0.00	1.19
2079	37	2^{29}	500	0.04	1.44	2.98	1.33
3505	57	2^{40}	1000	0.08	1.82	0.21	1.74
9429	215	2^{165}	1000	0.00	4.31	0.01	4.16
61345	1434	2^{892}	2000	0.00	31.45	0.99	29.44
213521	4983	2^{3073}	5000	0.02	104.61	0.00	99.37
429586	10002	2^{7455}	10000	0.00	183.75	0.10	172.52

[a] 648 s [b] 2700 s [c] 639 s [d] 2645 s

Figure 9: A comparison of the performance of the brute-force approach and our inference algorithm on large expressions containing seeded type errors. The errors are either spread evenly or clustered within the expression. Our algorithm caught 100% of the errors in all cases, so we show only the time taken to do so. The running time of the brute-force approach was capped at one hour (3600 s). For cases that completed before this cap was reached, we give the running time as a footnote. For cases that did not complete, we ran each test 10 times starting from a random variant, and averaged the results.

we expect real software to be considerably less complex (from a variational perspective) than the expressions examined in this section, and very unlikely to induce worst-case scenarios. For example, some real-world studies have suggested an average choice nesting depth of just 1.5 [13]. However, it is possible that variational complexity is artificially limited by the inadequacy of current tools, which this work directly addresses.

10. Related Work

The work presented here builds on our previous work on typing variational programs [5]. In that work, we focused on establishing VLC as a foundation for work on typing variational programs (and other static analyses), introducing the notion of variational types, and developing the fundamentals of variational typing and variational type inference. In order to be practically useful, however, variational typing must support many more features. In [5] we demonstrated how this approach can be extended to support simple typing features like sum types. The work presented in this paper represents a much more significant and challenging extension to make the type system error-tolerant. This feature is critically important for typing real-world variational programs because it directly supports tasks like error location and incremental development.

In general, our approach distinguishes itself from related work in the field of SPLs by representing variation more generally and at a finer granularity, and by solving the more general problem of type inference rather than the type- or definedness-checking of explicitly typed programs [11, 12, 25].

Choice types are similar to variant types [10], which are used to uniformly manipulate 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 does not require such annotations.

Choice types are also reminiscent of union types [7]. A union type is an agglomeration of simpler types. For example, a function `f` might accept the union of types `Int` and `Bool`. Function application is then well typed if the argument's type is an element of the union type (either `Int` or `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.

Although they share a name, our notion of partial types differs from the work of Thatte [26]. Thatte's partial types provide a way to type certain objects that are not typable with simple types in lambda calculus, such as heterogeneous lists and persistent data. They are more similar to our typing patterns. Thatte's "untyped" type Ω represents an arbitrary well-typed expression, similar to our $\top$ type, while his inclusion relationship on partial types ($\leq$) is similar to our more-defined relationship on patterns ($\trianglelefteq$). Type inference with Thatte's partial types was proved decidable [14, 17], a property that holds for our type system also.

Top and bottom types in subtyping [19] are also similar to the types $\top$ and $\bot$ used in typing patterns. Moreover, the subtyping relationship plays a similar role to that of $\trianglelefteq$ on typing patterns. For example, all types are subtypes of the top type, which corresponds to the fact that all typing patterns are less or equally defined as $\top$ (similar for $\bot$ and the bottom type). However, the role of these type bounds is quite different. The top and bottom types are introduced to facilitate the proofs of certain properties and the design of type systems, for example, in bounded quantification [18], whereas the $\top$ and $\bot$ types are used as parts of larger patterns to track which variants are ill-typed, and to mask result types accordingly.

Our work is also related to the work of Siek et al. on gradual typing [22, 23]. The goal of that work is to integrate static and dynamic typing into a single type system. They use the symbol ? to represent a type that is not known statically (that is, it is a dynamic type). This is similar to our $\bot$ type in partial types and

typing patterns, particularly in the way it is used to determine a notion of *informativeness*. A type is less informative if it contains more ? types (or rather, if more of the type is subsumed by ? types). This relation is similar to an inverse of our more-defined relation on typing patterns, where a pattern becomes less defined as it is subsumed by $\perp$ types. The biggest difference between this work and our own is that $\perp$ types represent parts of a variational program that are statically known to be type incorrect, whereas the parts of a program annotated with ? types may still be dynamically type correct. Also, their system isolates ? types as much as possible with respect to a plain type, while we allow $\perp$ types to propagate outward in plain types, but contain $\perp$ types to as few *variants* as possible. This is best demonstrated by the fact that (if we extend the notion of definedness to partial types) $\perp$ is equally defined as $\perp \to \texttt{Int}$, but ? is strictly less informative than $? \to \texttt{Int}$.

Generating informative error messages and determining the causes and locations of type errors has been extensively studied in type systems [15, 28]. Our type system does not address this problem per se, as far as individual program variants is concerned. However, a partial type does indicate which variants contain type errors. This information can be combined with traditional, single-variant systems to improve error location in variational programs.

The unification problem for equational theories that contain distributivity and associativity is known to be undecidable [27]. However, it is decidable when an idempotency law is added [1]. Therefore, because of choice idempotency, our unification problem is decidable. As we have shown, our problem is also unitary. This is important for implementing the type inference algorithm because it is necessary for a type system that has the principal typing property.

11. Conclusion and Future Work

We have presented a type system and inference algorithm for assigning partial types to variational programs. We have shown that the addition of error types and the resulting more-defined ordering for types integrates well with a variational type system. Specifically, we were able to extend the unification and type inference algorithms to produce most-general partial types for variational lambda calculus expressions. These results are an important step toward providing type-system support for massively variational software, and for the incremental development of variational programs.

In future work we plan to explore incremental variational type inference. We expect that programmers often work on only a small subset of variants at a time, and so there is huge opportunity for efficiency gains by reusing the unchanged variational context in typing incremental changes. We expect this historical information to also be useful for producing more precise error feedback. We also plan to investigate how variational typing can be used to support strong but flexible typing in functional, staged programming languages.

Acknowledgments

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

References

[1] S. Anantharaman, P. Narendran, and M. Rusinowitch. Unification Modulo ACUI Plus Homomorphisms/Distributivity. *Journal of Automated Reasoning*, 33:1–28, 2004.

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

[3] F. Baader and W. Snyder. Unification Theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 8, pages 445–533. Elsevier Science Publishers, Amsterdam, NL, 2001.

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

[5] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. Technical Report, School of EECS, Oregon State University, 2012. Available at: `http://eecs.oregonstate.edu/~erwig/ToSC/VLC-TypeSystem.pdf`.

[6] L. Damas and R. Milner. Principal Type Schemes for Functional Programming Languages. In *9th ACM Symp. on Principles of Programming Languages*, pages 207–208, 1982.

[7] M. Dezani-Ciancaglini, S. Ghilezan, and B. Venneri. The "Relevance" of Intersection and Union Types. *Notre Dame Journal of Formal Logic*, 38(2):246–269, 1997.

[8] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. on Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

[9] GHC. The Glasgow Haskell Compiler. `http://haskell.org/ghc`.

[10] K. Kagawa. Polymorphic Variants in Haskell. In *ACM SIGPLAN Workshop on Haskell*, pages 37–47. ACM, 2006.

[11] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. on Software Engineering and Methodology*, 2012. To appear.

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

[13] C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Int. Conf. on Generative Programming and Component Engineering*, pages 19–23, 2008.

[14] D. Kozen, J. Palsberg, and M. I. Schwartzbach. Efficient Inference of Partial Types. In *Journal of Computer and System Sciences*, pages 363–371, 1992.

[15] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for Type-Error Messages. In *ACM Conf. on Programming Language Design and Implementation*, pages 425–434, 2007.

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

[17] P. M. O'Keefe and M. Wand. Type Inference for Partial Types is Decidable. In *European Symp. on Programming*, pages 408–417, 1992.

[18] B. C. Pierce. Bounded Quantification with Bottom. Technical report, Computer Science Department, Indiana University, 1997.

[19] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[20] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlang, Berlin Heidelberg, 2005.

[21] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[22] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[23] J. G. Siek and M. Vachharajani. Gradual Typing with Unification-Based Inference. In *Symp. on Dynamic Languages*, pages 7:1–7:12, 2008.

[24] W. Taha and T. Sheard. MetaML and Multi-Stage Programming with Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[25] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Int. Conf. on Generative Programming and Component Engineering*, pages 95–104, 2007.

[26] S. Thatte. Type Inference with Partial Types. In *Int. Colloq. on Automata, Languages and Programming*, pages 615–629, 1988.

[27] E. Tiden and S. Arnborg. Unification Problems with One-Sided Distributivity. *Journal of Symbolic Computation*, 3(1-2):183–202, 1987.

[28] M. Wand. Finding the Source of Type Errors. In *ACM Symp. on Principles of Programming Languages*, pages 38–43, 1986.