A Calculus for Variational Programming

Sheng Chen[1], Martin Erwig[2], and Eric Walkingshaw[2]

1 **University of Louisiana at Lafayette**
`chen@louisiana.edu`

2 **Oregon State University**
`{erwig,walkiner}@oregonstate.edu`

Abstract

Variation is ubiquitous in software. Many applications can benefit from making this variation explicit, then manipulating and computing with it directly—a technique we call "variational programming". This idea has been independently discovered in several application domains, such as efficiently analyzing and verifying software product lines, combining bounded and symbolic model-checking, and computing with alternative privacy profiles. Although these domains share similar core problems, and there are also many similarities in the solutions, there is no dedicated programming language support for variational programming. This makes the various implementations tedious, prone to errors, hard to maintain and reuse, and difficult to compare.

In this paper we present a calculus that forms the basis of a programming language with explicit support for representing, manipulating, and computing with variation in programs and data. We illustrate how such a language can simplify the implementation of variational programming tasks. We present the syntax and semantics of the core calculus, a sound type system, and a type inference algorithm that produces principal types.

1998 ACM Subject Classification F.3.3 Logics and Meanings of Programs, D.3.2 Programming Languages

Keywords and phrases Variational programming, variational types, variability-aware analyses

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

The idea of representing and computing with *explicit variation* has been used to solve a variety of problems where a similar computation must be performed on sets of related data. For example, a challenge in *software product lines* (SPL) is to ensure that *all* program variants that can be generated from a code base satisfy a given property. Since analyzing each variant in sequence is intractable, researchers have developed algorithms and tools for analyzing the variational code directly. This work enables the efficient parsing [29], type checking [28, 46, 2], type inference [17, 16], model checking [19, 1], and flow analysis [6, 7] of whole software product lines at once.

But the need to write programs that manipulate variation representations is not limited to the area of software product lines. Other applications include computing with alternative privacy policies [3], improving type error messages [10, 11, 15], improving the performance of simulations [45], supporting view-based editing [53], and several applications to software testing [30, 31, 39, 49].

All these applications require an explicit representation of variation in data along with operations for inspecting, transforming, and/or mapping computations over such variational data. We collectively refer to these as *variational programming* tasks. Variational programming is not well supported by existing programming languages. While there exists some support from libraries [24, 29], there is no dedicated language support for variational programming. The lack of language support makes it difficult to reason about the variational programs and the artifacts they produce.

Therefore, we present in this paper a set of carefully designed, dedicated variational programming abstractions that can provide guarantees about variational programs and their results. The integration of these operations into a functional programming language can greatly improve the effectiveness, reliability, and productivity of variational programming.

In the remainder of this paper, we first motivate variational programming and illustrate it with several examples. We then formalize the underlying principles by defining a calculus for variational programming. This paper makes the following contributions.

- We explain the idea of variational programming through several examples in Section 2. We introduce the underlying variation representation and demonstrate how variational programming combinators can be built from a small set of core constructs and principles.
- We present the syntax and operational semantics of a variational programming calculus (VPC) in Section 3. A particular challenge is to balance and integrate the competing notions of variation-preserving computation and variation aggregation. A careful choice of congruence rules and evaluation strategy ensures the confluence of the semantics.
- We develop a type system for VPC in Section 4 and present a soundness theorem. Named choices lead to a limited form of dependent types, called *dimension polymorphism*, which reflect variability in expressions at the type level. Moreover, since operations for the elimination of variation cause subtle changes in variational types, we have to employ constraints in the type system to track the evolution of variability in computations.
- In Sections 5 and 6, we present constraint generation and constraint solving that constitute a type inference algorithm for VPC, which is sound, complete, and principal.

We discuss related work in Section 7 and conclude the paper with Section 8.

2 Variational Programming

The term “variational program” has two related meanings. On the one hand, it refers to a program family from which individual programs can be selected. On the other hand, it refers to programs that process variational values. For variational programs that process variational programs the two meanings collapse. In this paper we are primarily focused on the second meaning, that is, we want to study programs that process variational values.

The systematic processing of variational values requires machinery to create, query, and transform such values. These tasks should be supported by a principled variation representation that enjoys a rich set of laws. In Section 2.1 we present a simple representation based on the choice calculus [23], illustrate how it can be embedded into a (functional) programming language, and then motivate and outline the rest of this section.

2.1 Representing Variation

We represent variation by named, binary choices [23]. For example, a program that applies a function `f` to either `1` or `True` can be represented as `p = \f.f A<1,True>` where the choice represents a point of variation within `p`. A program that contains choices is called a *variational program*. This variational program encodes two plain programs, `\f.f 1` and `\f.f True`, called *variants*. These variants can be obtained by *selecting* either the left or right alternative from the choice. The name associated with the choice, A, is called a *dimension*. Dimensions allow the synchronization of choices in different parts of a program. For example, the program `q = p A<succ,not>` runs `p` with a choice of functions that will be synchronized with the choice of constants. Variational programs can vary in more than one dimension. Choices in different dimensions vary independently of one another, that is, whereas `q` contains only two variants, the variational program `q' = p B<succ,not>` contains four.

Executing a variational program produces a variational result. For example, executing program `q` above produces the result `A<2,False>`. This result may, in general, vary in all of the dimensions referred to in the variational program. We call such executions *variation-preserving* (see Section 2.2).

However, variational programming encompasses more than just executing variational programs. For example, suppose we have implemented a variational function that computes the time needed to complete a project depending on many design decision each represented by a different dimension. We can imagine several subsequent operations to perform on the variational result of this function. For example, we may want to do further variation-preserving computations by adding the variational time to complete the project to the variational time needed to complete another project, or by comparing it with a variational time computed by a different function. We may also want to *filter* the result by making selections that reduce the amount of variability in the result, which can help speed up subsequent variation-preserving computations. Alternatively, we may want to *aggregate* the variation, for example, to identify the minimum or maximum variant, to determine how to proceed depending on if we are being paid on salary or hourly. In the rest of this section, we take a closer look at variation-preserving computations (Section 2.2), reducing variation through selection (Section 2.3), and variation aggregation (Section 2.4), introducing the core constructs of our language and deriving some general combinators to support variational programming, as illustrated through an example application (Section 2.5). Finally, in Section 2.6, we argue why our language design is a good basis for variational programming.

2.2 Variation-Preserving Computations

We start with the task of adding variational numbers. Since choices in the same dimension are synchronized, expect evaluating `A<1,2> + A<3,4>` to produce `A<4,6>`. But how about `A<1,2> + B<3,4>` or even `A<1,2> + 3`? We can address the latter expression by realizing that a non-variational number is the same as a choice whose alternatives are identical, that is, `3 = A<3,3>`. In other words, choices are *idempotent*. Therefore, `A<1,2> + 3 = A<1,2> + A<3,3> = A<4,5>`. We arrive at the same result by considering that the variational expression `A<1,2> + 3` represents two plain expressions `1+3` and `2+3`, depending on which alternative from dimension `A` is selected, that is, the expression is equivalent to `A<1+3,2+3>`, which evaluates to `A<4,5>`.

We can see that the operation `+3` was effectively "moved" into the `A` choice, which is an application of the *choice distribution* law [23] that allows one to push a common context into a choice. This is illustrated below where C is an arbitrary context and D is an arbitrary dimension.

$$C[D\langle e, e'\rangle] = D\langle C[e], C[e']\rangle$$

Choice distribution is the basis for variation-preserving computation. It says that we can evaluate an expression such as `f A<3,4>` by applying `f` to the alternatives of the `A` choice. With this we can also determine the result of `A<1,2> + B<3,4>` by moving one of the choices into the other and then doing this again in each of the alternatives.

```
A<1,2>+B<3,4> = A<1+B<3,4>,2+B<3,4>> = A<B<1+3,1+4>,B<2+3,2+4>> = A<B<4,5>,B<5,6>>
```

If we decide to move the `A` choice instead, we obtain the equivalent value `B<A<4,5>,A<5,6>>`. This can be verified by making all possible selections for `A` and `B`.

Previously, we argued that `A<1,2> + A<3,4>` yields only two variants since both `A` choices are synchronized. It is instructive to perform choice distribution on this example to illustrate that selection commutes with semantics-preserving computation. In fact, we obtain the same derivation except that all `B`s are replaced by `A`s, so the resulting expression is `A<A<4,5>,A<5,6>>`. Observe that both `5` values can never be selected since selecting either the left or right alternative of `A` implies the same selection in the nested `A` choices. This phenomenon, called *choice domination*, neutralizes the effect of nested choices in the same dimension on the variability of values. These examples illustrate that we can apply a (variational) function to a (variational) value, and choice distribution will produce a result in which the variation in both the function and argument are preserved.

As another example, we can also compare variational values by a variation-preserving computation. For example, `A<3,5> == A<4,5>` evaluates to `A<False,True>`, and `A<B<5,4>,7>` ≤ `A<6,C<8,2>` yields `A<True,C<True,False>>`.

2.3 Eliminating Variation

The elimination of choices is called *selection*. Selection takes a selector of the form $D.s$ (where D is a dimension and s is either `L` or `R`), traverses the expression, and replaces all choices named D with its corresponding left or right alternative. For example, given program `q` defined in Section 2.1, the expression `sel A.L q` produces the variant `(\f.f 1) succ`. Since the program `q'` contains choices in two different dimensions, we need two selection operations to eliminate the variability in the program and obtain plain expressions. By considering all four possible selections on `q'`, we can observe that only two of the variants are type correct.

```
sel A.L (sel B.L q') = (\f.f 1) succ      sel A.R (sel B.L q') = (\f.f True) succ
sel A.L (sel B.R q') = (\f.f 1) not       sel A.R (sel B.R q') = (\f.f True) not
```

A choice is an expression, not a textual macro.[1] This means that we cannot vary, say, a few letters in the middle of an identifier or optionally exclude a closing parenthesis. However, we can pass choices to functions, and observe and manipulate them at runtime.

2.4 Variation Aggregation

Besides selecting individual variants, we often want to systematically eliminate all the variability in a variational program. For example, to compute the minimum variant of a variational number, we have to eliminate all of the choices, replacing each one by the smaller of its two alternatives. This kind of aggregation is supported by the syntactic form **any** d **from** e **in** e' **else** e''. An **any**-expression checks if there are any choices in expression e. If so, it picks one out, binds the corresponding dimension name to a new variable d, then returns expression e', which may contain references to d. If no choices exist in e, the **any**-expression evaluates to e''. The following function `vmin` uses **any** to implement the minimum-variant aggregation. If a choice is found in its argument `i`, it eliminates the choice with `sel` and recursively computes the minimum variant of the left and right alternatives, then returns the minimum of those two values (using the function `min`). If no choice is found, then all variation has been eliminated and `i` is a plain integer, which is simply returned.

[1] This is in contrast with, for example, the C Preprocessor's `#ifdef` notation, which is widely used to implement static variation in C programs.

```
vmin : Int -> Int
vmin i = any d from i in
             min (vmin (sel d.L i)) (vmin (sel d.R i))
         else i
```

Now consider the expression `vmin A<B<4,3>,2>`, which we expect to evaluate to `2`. However, by applying the choice distribution principle from the previous subsection, the expression could also evaluate to `A<B<4,3>,2>`. In other words, in this scenario, choice distribution prevents aggregation and changes the intended semantics of the program. There are several ways to address this issue, but we choose to disambiguate the situation syntactically, which leads to a simpler operational semantics. Specifically, we allow function arguments to be annotated as "aggregating" (using `@`), which acts as a boundary that prevents choice distribution. In the case of `vmin` this looks as follows.

```
vmin @i = any d from i in ...
```

Now `vmin A<B<4,3>,2>` will compute `2` and not return a choice. But what if we want to independently compute `vmin` for two alternatives of a dimension, say `A`? We can still do this by applying a corresponding choice of `vmin`s, that is, we write `A<vmin,vmin> A<B<4,3>,2>`. Now choice distribution will preserve the `A` dimension and we get the result `A<3,2>`. However, note that the idempotency law does not hold for variation aggregators, that is, in general we have `A<vmin,vmin> ≠ vmin`.

Within the definition of `vmin` are two inter-related patterns that occur frequently in variational programs. The first is to select both the left and right alternatives of the same dimension in parallel; the second is to pick an arbitrary dimension, then immediately select with it. To accommodate the first pattern, we introduce a derived form, **split**, which allows us to rewrite `vmin` as follows.

```
vmin @i = any d from i in
            split i on d<l,r> -> min (vmin l) (vmin r)
          else i
```

In the **split** expression, `d` refers to the dimension we want to eliminate, bound by the enclosing **any** expression; `l` and `r` are new names bound to the result `sel d.L i` and `sel d.R i`, respectively.

To accommodate the second pattern, we combine **split** and **any** into a single construct as illustrated below. This expression form picks a dimension out of `i` and immediately splits on it.

```
vmin @i = split i on any
                d<l,r> -> min (vmin l) (vmin r)
           else i
```

Which dimension is picked by **any** is determined by a fixed, predefined ordering among dimensions. Therefore the names of dimensions in a variational value generally matter. However, in functions that systematically process all variability present in a value, the order does not matter. For example, `vmin A<B<5,4>,7>` will compute the same result, independent of the order in which the dimensions are picked by **any**. To demonstrate, here is the computation that unfolds if `A` is picked first.

```
min (vmin B<5,4>) (vmin 7) = min (min 5 4) 7 = min 4 7 = 4
```

If `B` is picked first, the enclosing `A` choice is preserved across both alternatives of `B`.

```
min (vmin A<5,7>) (vmin A<4,7>) = min (min 5 7) (min 4 7) = min 5 4 = 4
```

The two computations return the same result because choices commute [23] allowing arbitrary reordering of how choices are nested. Specifically, for any two dimensions, `A` and `B`, the following relationship holds.

```
A<B<a,b>,B<c,d>>  ≡  B<A<a,c>,A<b,d>>
```

Computing the minimum variant of a variational integer is an instance of the more general task of aggregating variability, which can be captured by a variational join function that accumulates variants with a binary function.

```
vjoin : (a -> a -> a) -> a -> a
vjoin f @x = split x on any
                   d<l,r> -> f (vjoin f l) (vjoin f r)
              else x
```

Using `vjoin` we can define `vmin` more simply as `vjoin min`. More generally, we can define a variational fold that both converts each variant to a separate accumulator type `b`, then aggregates them.

```
vfold : (a -> b) -> (b -> b -> b) -> a -> b
vfold f g @x = split x on any
                     d<l,r> -> g (vfold f g l) (vfold f g r)
                else f x
```

Now `vjoin` is just a `vfold` where the conversion is the identity function, that is, `vjoin = vfold id`.

Using `vfold`, we can aggregate variational values in a variety of ways. For example, here is a function for counting the number of choices in a variational value.

```
choices : a -> Int
choices = vfold (\x.0) (\x y.1+x+y)
```

The `vfold` function is essentially a bottom-up tree fold over the binary trees of choices. However, because of the semantics of **any**, the aggregation order is determined not by the structure of the value, but by the ordering of dimension names contained in the value. This means that one should be careful when using `vfold` or `vjoin` with aggregating functions that are not associative.

2.5 An Application to Variational Unification

Variational unification finds substitutions that solve equations of the form: $A\langle \texttt{Int}, a\rangle \equiv^? B\langle b, c\rangle$. This is not a trivial problem. The following substitution is an obvious solution to this example.

$$\sigma_1 = \{a \mapsto \texttt{Int},\ b \mapsto \texttt{Int},\ c \mapsto \texttt{Int}\}$$

However, while some simple unifiers such as σ_1 can be found quickly, identifying the most general unifier is not easy. To wit, here is a list of some other solutions, some more general than others, some unrelated. Substitution σ_6 is the most general unifier.

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

One particular challenge for the unification algorithm is to find substitutions modulo an equivalence relation ($\equiv$) that includes laws for choice distribution, idempotency, and domination, which have been described already, plus some others.

A variational unification algorithm was first presented in [17]. This algorithm has since been employed as a crucial component in a number of applications [17, 16, 10, 12, 13, 14]. The original implementation in Haskell required a considerable number of auxiliary type and function definitions to support handling choice types. Having choices available as part of the language simplifies the implementation tremendously, allowing the programmer to focus on the main logic. First, the data type definition for types does not need to mention choice types and only needs to provide the (non-variational) constructors to represent the original (non-variational) core type language.

```
data Type = TInt | TBool | TVar Int | TFun Type Type
```

The unification of choice types works by systematically matching dimensions and type constructors, constructing substitutions along the way. The matching process is dominated by the matching of dimensions since they can be moved up or down in expressions (due to choice distribution/factoring), which is directly supported by the operation `split`. This is reflected in the following definition of the function `vunify`. Without going into too much detail, we explain how some of the cases work, and in particular, how they can profitably exploit the fact that variation is built into the language.

```
vunify : Type -> Type -> Subst
vunify @t @u = split t on any d<lt,rt> ->
                 split u on d<lu,ru> -> d<vunify lt lu,vunify rt ru>         (1)
                 else                   d<vunify lt  u,vunify rt  u>         (2)
               else
                 split u on any d<lu,ru> -> d<vunify t lu,vunify t ru>       (3)
                 else                       unify t u                        (4)
```

In case (1), if both types contain the same dimension, `vunify` computes a choice of substitutions for both alternatives, which is equivalent to a substitution with choice types.

$$D\langle\{a \mapsto T, \ldots\}, \{a \mapsto T', \ldots\}\rangle = \{a \mapsto D\langle T, T'\rangle, \ldots\}$$

Since it can happen that either substitution contains a mapping for a variable that isn't contained in the other, choice factoring can be "blocked" in such cases. We can write a function that explicitly converts a variational substitution into a substitution containing choice types and that takes care of these cases by adding a type variable for missing substitutions; we omit the code here for brevity.

In case (2), if `t` contains some dimension `d` but `u` does not, we can unify each of `t`s alternatives with `u` due to idempotency (that is, `u` = `D<u,u>`). The case (3) is symmetric. Finally, in case (4), neither type contains any more dimensions, so we invoke the non-variational function `unify`, which implements Robinson's traditional unification algorithm. This example illustrates how variational concerns can be isolated from the rest of the unification algorithm; `vunify` focuses on variation, while `unify` handles the non-variational type structure.

2.6 A Foundational Language for Variational Programming

We argue that the features described in this section constitute a rational basis for a variational programming language. A choice between two alternatives is by definition the simplest possible representation of variation, and the choice calculus has a well-developed theory [23, 50] that has been successfully reused in several contexts [17, 16, 10, 12, 13, 14, 24, 52, 53, 11, 15]. Choices are also used internally in the TypeChef system [29], which has been reused in a number of projects for analyzing software product lines [37, 38, 33, 30, 34], and choices are the basic representation used in other variational programming scenarios outside of software product lines [3, 48]. Choices encode variation *in-place* but can also be combined with compositional approaches to variation [51]. Choices between several alternatives can be modeled by cascading choices in different dimensions. In previous work [53], we have shown how the choice calculus can be extended with more general conditions on choices, but we omit this extension here to keep the presentation concise.

The concept of variation-preserving computation describes the property that selection commutes with evaluation. That is, running a variational program corresponds to running all of the individual variants separately. This is the philosophy expressed in almost all work on analyzing software product lines [47] and in recent work on variational execution [39, 3]. Selection is a symmetric elimination form for choices that satisfies Gentzen's principle [25, p. 102]. This means that choice and selection are information-preserving inverses, a property that enables simple and reversible semantics.

Finally, a necessary feature for practical programming with variations is *reflection* on variability. In Section 2.4, we describe the **any** form for extracting one arbitrary dimension if it exists, and in the next section we describe a similar **the** form that extracts one specific dimension if it exists. The **any** and **the** forms are orthogonal constructs that are sufficient to implement pattern matching on variations, as illustrated by the derived form **split**.

3 Syntax and Semantics of VPC

In this section we define a variational programming calculus (VPC). The calculus extends the lambda calculus with all of the features needed to implement the variational functional programming language described in the previous section. We define the syntax of VPC in Section 3.1 and a small-step operational semantics in Section 3.2.

3.1 Syntax

We separate the definition of VPC into two parts: (1) a core calculus that includes only the essential features of the language, and (2) several derived forms, defined in terms of the core calculus, that provide convenient surface syntax for variational programming.

Core calculus.

The syntax of the core calculus is defined in Figure 1. The first two lines define separate namespaces for variables that refer to expressions and variables that refer to dimension names. In this paper, we distinguish these namespaces by using letters from the beginning of the alphabet for dimension variables, and from the end of alphabet for expression variables. Literal dimension names are represented by capital letters. The syntactic category δ includes both dimension variables and dimension names, while φ includes both dimension variables and expression variables. As a convention, we use the nonterminal of a syntactic category (or variations on it) to stand for arbitrary instances of that category. For example, φ, φ_1, and φ_2 all generically refer to variables where it is not important to distinguish between expression variables and dimension variables. We explain the syntactic categories s and $\star$ in the context of the relevant expression forms.

$$
\begin{array}{rcll}
v & ::= & x \mid y \mid z \mid \ldots & (\textit{expression variables}) \\
d & ::= & a \mid b \mid c \mid \ldots & (\textit{dimension variables}) \\
D & ::= & A \mid B \mid C \mid \ldots & (\textit{dimension names}) \\
\delta & ::= & d \mid D & (\textit{dimensions}) \\
\varphi & ::= & v \mid d & (\textit{variables}) \\
s & ::= & \ell \mid r & (\textit{selectors}) \\
* & ::= & @ \mid \epsilon & (\textit{abstraction annotations}) \\
\\
e & ::= & \kappa \mid \lambda^{*}\varphi.e \mid e\ e \mid v & (\textit{lambda calculus + constants}) \\
 & \mid & \textbf{let}\ v = e\ \textbf{in}\ e & (\textit{recursive let}) \\
 & \mid & \delta & (\textit{dimensions}) \\
 & \mid & \delta\langle e, e\rangle & (\textit{choice}) \\
 & \mid & \delta_s \blacktriangleright e & (\textit{selection}) \\
 & \mid & \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e\ \textbf{else}\ e & (\textit{any dimension pick}) \\
 & \mid & \textbf{the}\ \delta\ \textbf{from}\ e\ \textbf{in}\ e\ \textbf{else}\ e & (\textit{specific dimension pick}) \\
\\
\underline{e} & ::= & \kappa \mid D \mid \lambda^{*}\varphi.e \mid D\langle \underline{e}, \underline{e}\rangle & (\textit{values})
\end{array}
$$

Figure 1 Syntax of core VPC.

The metavariable e ranges over VPC expressions. The first two lines enumerate the constructs of the lambda calculus extended by constants, ranged over by κ, and recursive **let**-expressions. Abstractions differ from standard lambda calculus in two ways. First, they can be distinguished by the namespace of their declared variable. That is, we will sometimes distinguish between expression abstractions, $\lambda v.e$, and dimension abstractions, $\lambda d.e$. Second, they may be optionally annotated by an @ symbol, as in $\lambda^{@}\varphi.e$, which prevents mapping the function over variation in its argument, as described in Section 2.4. Note that the alternative annotation ϵ represents the *lack* of an @ annotation; that is, we write simply $\lambda\varphi.e$ for an unannotated abstraction. In addition to constants and expression variables, the third line defines that we can also refer to dimension literals and dimension variables.

The next two lines define the choice and select constructs, described in Section 2.1 and Section 2.3, for introducing and eliminating variation, respectively. The dimension δ associated with a choice $\delta\langle e_1, e_2\rangle$ may be either a literal dimension name D or a reference to a dimension variable d. A selection replaces, within e, all choices in dimension δ by either their left alternatives, written $\delta_\ell \blacktriangleright e$, or their right alternatives, written $\delta_r \blacktriangleright e$.

The **any** d **from** e **in** e' **else** e'' and **the** δ **from** e **in** e' **else** e'' forms match against the variability in e. The **any** form is described in Section 2.4, while **the** is new. For **any**, if e contains variation, then d will be bound to a dimension from e and e' will be evaluated, otherwise d will remain undefined and e'' will be evaluated. For **the**, if e contains variation in the specified dimension δ, then e' will be evaluated, otherwise e'' will be evaluated. Note that **any** *declares* a new dimension variable d while **the** *refers* to an existing dimension name or variable δ.

Values.

The last line in Figure 1 describes the *values* of VPC as a subset of expressions. Values are produced by fully evaluating well-typed and terminating VPC expressions. In addition to

$$\begin{array}{ll}
\textbf{split}\ e\ \textbf{on}\ \delta\langle v_\ell, v_r\rangle \rightarrow e' & \rightsquigarrow (\lambda v_\ell\, v_r.e')\ (\delta_\ell \blacktriangleright e)\ (\delta_r \blacktriangleright e) \\
\textbf{split}\ e\ \textbf{on}\ \delta\langle v_\ell, v_r\rangle \rightarrow e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{the}\ \delta\ \textbf{from}\ e\ \textbf{in}\ \textbf{split}\ e\ \textbf{on}\ \delta\langle v_\ell, v_r\rangle \rightarrow e'\ \textbf{else}\ e'' \\
\textbf{split}\ e\ \textbf{on}\ \textbf{any}\ d\langle v_\ell, v_r\rangle \rightarrow e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ \textbf{split}\ e\ \textbf{on}\ d\langle v_\ell, v_r\rangle \rightarrow e'\ \textbf{else}\ e'' \\
\textbf{ifvar}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'' \\
\textbf{ifplain}\ e\ \textbf{then}\ e'\ \textbf{else}\ e'' & \rightsquigarrow \textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e''\ \textbf{else}\ e'
\end{array}$$

Figure 2 Derived syntactic forms for VPC.

constants and abstractions, which are typical values in lambda calculus, a value can also be dimension literal or a choice whose alternatives are also values. Note that only choices with literal dimension names may be values. This implies that expressions with free dimension variables are not well-typed (see Section 4). Additionally, observe that the body of an abstraction value may be an arbitrary expression.

Derived forms.

Figure 2 extends the syntax of VPC with several derived forms that macro-expand to constructs in the core calculus. These forms enable higher-level operations on variational values without complicating the semantics and type system of VPC.

The first group of derived forms are three variations on the **split** construct described in Section 2.4. A basic **split** expression makes both possible selections on e in dimension δ and binds these to v_ℓ and v_r in e' using function application. The **split**–**else** form expands to a **the** expression that checks whether δ appears in a free choice in e; if so, it performs the split, otherwise it returns e''. Finally, the **split**–**any**–**else** form expands to an **any** expression that picks an arbitrary dimension out of e, if one exists, then either splits on that dimension or returns e'' if there is no variation in e.

The second group introduces two derived forms for basic queries about the presence (**ifvar**) or non-presence (**ifplain**) of variability in e. These expand to **any**-expressions that declare an arbitrary fresh dimension variable d, which is not referenced in e'.

3.2 Operational Semantics

In this section we define a small-step operational semantics for VPC. Since derived forms macro-expand into the core calculus before evaluation, we consider only the core calculus here. The step relation has the form $e \longrightarrow e'$ and is defined in Figure 3. We separate the discussion of the rules into three parts: (1) reduction rules, which form the core of the semantics, (2) commutation rules for setting up future reductions, and (3) congruence rules for enabling reduction in subexpressions.

Reduction rules.

The most basic variation constructs are choice and selection, for introducing and eliminating variation. The elimination of choices by selections is defined by the two CHC-ELIM rules, for selecting the left or right alternative of a choice. Observe that selection is only defined on choices with literal dimension names, not dimension variables. This requires that all dimension variables be substituted (see APP-REDUCE) before choices can be eliminated. Also note that these reductions propagate the selection into the remaining alternative, implementing *choice*

Chc-Elim-L
$$D_\ell \blacktriangleright D\langle e_1, e_2\rangle \longrightarrow D_\ell \blacktriangleright e_1$$

Chc-Elim-R
$$D_r \blacktriangleright D\langle e_1, e_2\rangle \longrightarrow D_r \blacktriangleright e_2$$

Sel-Idemp-K
$$\delta_s \blacktriangleright \kappa \longrightarrow \kappa$$

Sel-Idemp-D
$$\delta_s \blacktriangleright D \longrightarrow D$$

App-Reduce
$$(\lambda\varphi.e)\ e' \longrightarrow \textbf{split}\ e'\ \textbf{on any}\ d\langle v_\ell, v_r\rangle \rightarrow d\langle(\lambda\varphi.e)\ v_\ell, (\lambda\varphi.e)\ v_r\rangle\ \textbf{else}\ [e'/\varphi]e$$

App$^{@}$-Reduce
$$(\text{ƛ}^{@}\varphi.e)\ e' \longrightarrow [e'/\varphi]e$$

Any-Reduce-Then
$$\textbf{any}\ d\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow [\min(\mathit{dims}(\underline{e}))/d]e'$$

Any-Reduce-Else
$$\varnothing = \mathit{dims}(\underline{e}) \Rightarrow \textbf{any}\ d\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow e''$$

The-Reduce-Then
$$\delta \in \mathit{dims}(\underline{e}) \Rightarrow \textbf{the}\ \delta\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow e'$$

The-Reduce-Else
$$\delta \notin \mathit{dims}(\underline{e}) \Rightarrow \textbf{the}\ \delta\ \textbf{from}\ \underline{e}\ \textbf{in}\ e'\ \textbf{else}\ e'' \longrightarrow e''$$

Let-Reduce
$$\textbf{let}\ v = e\ \textbf{in}\ e' \longrightarrow [[\textbf{let}\ v' = e\ \textbf{in}\ v'/v]e/v]e'$$

Sel-Let
$$\delta_s \blacktriangleright \textbf{let}\ v = e\ \textbf{in}\ e' \longrightarrow \textbf{let}\ v = \delta_s \blacktriangleright e\ \textbf{in}\ \delta_s \blacktriangleright e'$$

App-Chc
$$\delta\langle e_1, e_2\rangle\ e' \longrightarrow \delta\langle e_1\ (\delta_\ell \blacktriangleright e'), e_2\ (\delta_r \blacktriangleright e')\rangle$$

Sel-App
$$\delta_s \blacktriangleright e\ e' \longrightarrow (\delta_s \blacktriangleright e)\ (\delta_s \blacktriangleright e')$$

Sel-Abs-Exp
$$\delta_s \blacktriangleright \text{ƛ}^{*}v.e \longrightarrow \text{ƛ}^{*}v.(\delta_s \blacktriangleright e)$$

Sel-Abs-Dim
$$d' \text{ is fresh} \Rightarrow \delta_s \blacktriangleright \text{ƛ}^{*}d.e \longrightarrow \text{ƛ}^{*}d'.(\delta_s \blacktriangleright [d'/d]e)$$

Sel-Chc
$$D \neq D' \Rightarrow D_s \blacktriangleright D'\langle e_1, e_2\rangle \longrightarrow D'\langle D_s \blacktriangleright e_1, D_s \blacktriangleright e_2\rangle$$

Sel-Any
$$d' \text{ is fresh} \Rightarrow \delta_s \blacktriangleright (\textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') \longrightarrow \textbf{any}\ d\ \textbf{from}\ \delta_s \blacktriangleright e\ \textbf{in}\ \delta_s \blacktriangleright [d'/d]e'\ \textbf{else}\ \delta_s \blacktriangleright e''$$

Sel-The
$$\delta_s \blacktriangleright (\textbf{the}\ \delta'\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') \longrightarrow \textbf{the}\ \delta'\ \textbf{from}\ \delta_s \blacktriangleright e\ \textbf{in}\ \delta_s \blacktriangleright e'\ \textbf{else}\ \delta_s \blacktriangleright e''$$

Figure 3 Small-step operational semantics of VPC: reduction and commutation rules.

domination (see Section 2.2), illustrated by the following reduction sequence.

$$A_r \blacktriangleright A\langle \mathtt{2}, A\langle \mathtt{3}, \mathtt{4}\rangle\rangle \longrightarrow A_r \blacktriangleright A\langle \mathtt{3}, \mathtt{4}\rangle \longrightarrow A_r \blacktriangleright \mathtt{4} \longrightarrow \mathtt{4}$$

The final step of this sequence depends on the fact that selection on constants (and dimension names) is idempotent, as defined by the two Sel-Idemp rules. Our strategy for eliminating selections will be to continually push selections down to the leaves of an expression, eliminating choices as we go. This will be accomplished mainly by the commutation rules discussed later. However, note that we do not have a selection idempotency rule for selections applied to variables. Such expressions are stuck until a corresponding substitution enables further progress.

The next two rules define function application with aggregating (@-annotated) and non-aggregating functions. The App$^{@}$-Reduce rule is just standard β-reduction. For non-aggregating functions, recall from Section 2.2 that we must instead map the function application over all variants of the argument. In fact, both the LHS and RHS of an application may be variational. If the LHS is a choice, we can apply the App-Chc rule to push the application into both alternatives (and selecting on the RHS to enforce choice domination). We might expect a symmetric rule for pushing an application into a choice on the RHS of an application and then a β-reduction rule for reducing abstractions applied to non-choices. However, this strategy only maps across variation introduced by top-level choices, whereas in general variation may be arbitrarily embedded within an expression, such as in an abstraction body. Therefore, in App-Reduce we define that an unannotated redex is expanded into a **split-any-else** expression that recursively splits e' on every dimension it contains, effectively pushing the application downward into its variants. After we've split on all of the dimensions in e', the **else** branch applies, and the generated expression reduces to standard β-reduction.

Reducing an **any**- or **the**-expression requires querying the variability of the first subexpression, which we call the *scrutinee*. Observe that all of the Any and The reduction rules require the scrutinee to be a value. This forces full reduction of the scrutinee before the expression can be reduced. Ideally, we would reduce the scrutinee to either a choice or a constant, then replace the **any**/**the**-expression by its **then** or **else** branch, respectively. However, this strategy is stymied by the fact that abstractions are also a value form, and abstraction bodies are expressions that may contain choices, but may also contain free variables and so cannot be further reduced to values. This leaves two possible solutions: (1) get stuck when the scrutinee reduces to an abstraction, (2) statically approximate the variability of the reduced scrutinee. Since we want to match against arbitrary values, we choose option (2).

Figure 4 defines an auxiliary function $\mathit{dims}(e)$ that computes a set of dimensions that conservatively approximates the free dimensions of variability in e. A *free* dimension is either a dimension name or a dimension variable that is not bound by an enclosing abstraction in e. Observe in the definition of *dims* that dimension declarations and choices contribute to the result set, while selections (which eliminate choices and therefore variability) subtract from it. The dimension abstraction and **any** forms, which declare and scope dimension variables, also subtract from the result set since their dimension variables are not free in e.

Using *dims*, we can define the reduction rules for **the** and **any**. The The-Reduce rules simply check to see whether the specified dimension is present in $\mathit{dims}(\underline{e})$ and reduce to either the **then** or **else** branch accordingly. An **any**-expression reduces to the **else** branch if $\mathit{dims}(\underline{e})$ is empty (Any-Reduce-Else), but if it's not empty we must pick a dimension to bind to d (in Any-Reduce-Then). The dimension we pick is based on an ordering relation ($<$) on

$$
\begin{array}{llll}
\mathit{dims}(\kappa) = \mathit{dims}(v) = \{\} & & \mathit{dims}(e\ e') & = \mathit{dims}(e) \cup \mathit{dims}(e') \\
\mathit{dims}(\delta) = \{\delta\} & & \mathit{dims}(\delta\langle e, e'\rangle) & = \mathit{dims}(e) \cup \mathit{dims}(e') \cup \{\delta\} \\
\mathit{dims}(\lambda^{*} v.e) = \mathit{dims}(e) & & \mathit{dims}(\delta_s \blacktriangleright e) & = \mathit{dims}(e) - \{\delta\} \\
\mathit{dims}(\lambda^{*} d.e) = \mathit{dims}(e) - \{d\} & & &
\end{array}
$$

$$
\mathit{dims}(\textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') = \begin{cases} \mathit{dims}(e') - \{d\} & \text{if } \mathit{dims}(e) \neq \varnothing \\ \mathit{dims}(e'') & \text{otherwise} \end{cases}
$$

$$
\mathit{dims}(\textbf{the}\ \delta\ \textbf{from}\ e\ \textbf{in}\ e'\ \textbf{else}\ e'') = \begin{cases} \mathit{dims}(e') & \text{if } \delta \in \mathit{dims}(e) \\ \mathit{dims}(e'') & \text{otherwise} \end{cases}
$$

Figure 4 Static approximation of dimensions in a VPC expression.

$$
\begin{array}{ll}
(\lambda v.e_1)\ D\langle e_2, e_3\rangle & \\
\longrightarrow \textbf{split}\ D\langle e_2, e_3\rangle\ \textbf{on any}\ d\langle v_\ell, v_r\rangle \to d\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle\ \textbf{else}\ [D\langle e_2, e_3\rangle / v]e_1 & \\
& \{\textsc{App-Reduce}\} \\
\longrightarrow \textbf{any}\ d\ \textbf{from}\ D\langle e_2, e_3\rangle\ \textbf{in split}\ D\langle e_2, e_3\rangle\ \textbf{on}\ d\langle v_\ell, v_r\rangle \to & \\
\qquad d\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle\ \textbf{else}\ [D\langle e_2, e_3\rangle / v]e_1 & \{\mathit{expand}\ \textbf{split}\text{-}\textbf{any}\text{-}\textbf{else}\} \\
\longrightarrow \textbf{split}\ D\langle e_2, e_3\rangle\ \textbf{on}\ D\langle v_\ell, v_r\rangle \to D\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle & \{\textsc{Any-Reduce-Then}\} \\
\longrightarrow (\lambda v_\ell\, v_r.\, D\langle(\lambda v.e_1)\ v_\ell, (\lambda v.e_1)\ v_r\rangle)\ (D_\ell \blacktriangleright D\langle e_2, e_3\rangle)\ (D_r \blacktriangleright D\langle e_2, e_3\rangle) & \\
& \{\mathit{expand}\ \textbf{split}\} \\
\longrightarrow (\lambda v_\ell\, v_r.\, D\langle(\lambda v.e_1)\ v_\ell, (\lambda v.e_1)\ v_r\rangle)\ e_2\ e_3 & \{\textsc{Chc-Elim-L}, \textsc{Chc-Elim-R}\}
\end{array}
$$

Figure 5 Example reduction of $(\lambda v.e_1)\ D\langle e_2, e_3\rangle$.

dimensions: for any dimension name D and dimension variable d', $D \prec d'$; when comparing two dimension names or two dimension variables, $\prec$ is the lexicographic ordering of their names. The function $\min(\overline{\delta})$ returns the minimum $\delta \in \overline{\delta}$ according to this ordering relation. (We use the overbar to denote lists.)

The requirement that the scrutinee of an **any**/**the**-expression be fully evaluated is needed to ensure confluence since *dims* is an approximation of the actual variation in e. To illustrate, consider the expression[2] $\textbf{ifvar}\ (\lambda^{@} x.2)\ A\langle \mathtt{3}, \mathtt{4}\rangle\ \textbf{then}\ \mathtt{5}\ \textbf{else}\ \mathtt{6}$. Forcing evaluation of the scrutinee first yields reduces the expression to $\textbf{ifvar}\ \mathtt{2}\ \textbf{then}\ \mathtt{5}\ \textbf{else}\ \mathtt{6}$ and then to `6` since $\mathit{dims}(\mathtt{2}) = \varnothing$. However, if we allowed also reducing **any**/**the** immediately, the expression reduces to `5` since $\mathit{dims}((\lambda^{@} x.\mathtt{2})\ A\langle \mathtt{3}, \mathtt{4}\rangle) = \{A\}$.

Finally, the Let-Reduce rule implements a recursive **let** by a nested substitution.

Selection commutation rules.

The App-Chc rule described in the previous section pushes applications into choices on the LHS in order to setup future reductions via the App-Reduce or App$^{@}$-Reduce. Similarly, the eight Sel-* rules push selections downward to setup the elimination of matching choices via the Chc-Elim rules, and eventually to reduction at the leaves via the Sel-Idemp rules.

The Sel-App and Sel-Let rules straightforwardly push selections into applications and **let**-expressions, respectively. The two Sel-Abs rules push selections into abstractions. In

[2] Recall that **ifvar** expands to an **any**–**then**–**else** expression.

these rules we distinguish between expression abstractions (where the variable is in the namespace v) and dimension abstractions (in namespace d). However, both rules apply to both annotated and unannotated abstractions. We make this explicit by writing $\lambda^{*}v.e$ and assume that the annotations are preserved by the transformation. Note that the Sel-Abs-Dim rule renames the bound dimension variable to preemptively avoid the capture of the selected dimension δ, which may also be a dimension variable.

The Sel-Chc rule pushes selections into choices. This rule can only be applied if both dimensions (the one being selected, and the one referenced by the choice) have been resolved to dimension names. Selection on a choice involving a dimension variable is stuck until dimension substitution enables further progress. To illustrate why this is necessary, consider the expression $d_\ell \blacktriangleright d'\langle e_1, e_2\rangle$. At first it may seem that this expression is equivalent to $d'\langle d_\ell \blacktriangleright e_1, d_\ell \blacktriangleright e_2\rangle$ since the dimension variables have different names. However, it might be that d and d' are substituted by the same dimension name, in which case we should have eliminated the choice rather than pushing the selection into it.

Finally, the Sel-Any and Sel-The rules push selections into **any**- and **the**-expressions. Since **any** binds dimension names, the Sel-Any rule renames to avoid dimension variable capture.

Congruence rules.

We relegate to Appendix A.2 the congruence rules, which complete the reduction relation. They are entirely straightforward. We give them names like In-Any-Then for reducing the **then**-branch of an **any**-expression, and In-Chc-L for reducing the left alternative of a choice. The most interesting feature of the congruence rules are two rules that are *omitted*—we do not define In-Abs or In-Sel rules for reducing the body of abstractions or selections, respectively. The reason these are omitted is to avoid prematurely reducing **any**- and **the**-expressions. When an **any**/**the** occurs in the body of an abstraction, future variable substitutions may affect the variability of the scrutinee, and so we should not reduce it until all free variables have been substituted. Conversely, when an **any**/**the** occurs in the body of a selection, the scrutinee may contain *more* variation than it will after the selections have been pushed all the way down. Thus, reducing within abstractions and selections would lead to semantics that is not confluent.

Confluence.

The main result for the operational semantics is that the reduction relation is confluent, captured in the following lemma. (For a proof sketch, see Appendix A.1).

▸ **Lemma 1** (Local confluence). $e \longrightarrow e' \wedge e \longrightarrow e'' \implies \exists e'''.\ e' \longrightarrow^{*} e''' \wedge e'' \longrightarrow^{*} e'''$

4 Type System

This section presents a type system for vpc. In Section 4.1 we define the syntax of the type language, and in Section 4.2 we define the typing relationship through a set of typing rules.

4.1 Syntax

Types are stratified into three layers (see Figure 6). First, we use τ to range over *plain types*, which don't contain variations. Plain types include type constants γ, type variables α, and function types. Second, we use ϕ to range over variational types. A choice type $\delta\langle\phi_1, \phi_2\rangle$

$$\begin{array}{lcll}
\tau & ::= & \gamma \mid \alpha \mid \tau \to \tau & (\textit{plain types}) \\
\phi & ::= & \tau \mid \delta\langle\phi,\phi\rangle \mid \phi \to \phi \mid \delta & (\textit{variational types}) \\
\sigma & ::= & \forall\overline{\alpha}\overline{d}.\phi \mid \forall\overline{\alpha}\overline{d}.C \Rightarrow \phi & (\textit{variational schemas}) \\
C & ::= & \phi \equiv \phi \mid C \wedge C \mid \delta\langle C,C\rangle \mid \exists\overline{\alpha}\overline{d}.C \mid \delta \in \phi \Rightarrow \delta{\downarrow}\phi & (\textit{constraints})
\end{array}$$

Figure 6 Type Syntax

allows us to represent type variation. By including dimensions δ, which are expressions, in type syntax we obtain a lightweight form of dependent types. Finally, we have type schemas, which are variational types universally quantified over type variables α and dimension variables d. We refer to type schemas of the form $\forall d.\phi$ as *dimension-polymorphic types*. Note that we don't explicitly represent choice types containing polymorphic types since they can be transformed to type schemas, as shown in [10].[3] With this type syntax we can characterize types for expressions manipulating dimensions and choices. Consider, for example, the function *poly* that takes a dimension and creates a choice in that dimension.

$$\begin{array}{l}
\textit{poly} : \forall d.d \to d\langle \texttt{Int}, \texttt{Bool}\rangle \\
\textit{poly} = \lambda d.d\langle 2, \texttt{True}\rangle
\end{array}$$

In this example, the type precisely describes the intention of *poly*. While we can instantiate *poly* with any dimension, we may want to restrict dimensions that can be used to instantiate dimension-polymorphic types in general. We use the following expression *bounded* to illustrate this point.

$$\textit{bounded} = \lambda^{@}e.\textbf{any}\ d\ \textbf{from}\ e\ \textbf{in}\ d\langle 2, \texttt{True}\rangle\ \textbf{else}\ \texttt{undefined}$$

Here *bounded* finds a dimension in the expression supplied as argument based on some ordering that is discussed in Section 3. What should be the type of *bounded*? The type $\forall d.\phi \to d\langle \texttt{Int}, \texttt{Bool}\rangle$ is not precise enough since it can be instantiated with any dimension, even one absent in the input expression. To address this issue, we introduce constrained types that allow us to attach constraints to polymorphic types. Constraints specify the valid ways of instantiating polymorphic variables. To represent the type for *bounded*, we introduce the constraint $d{\downarrow}\phi$, which requires that d must be the particular dimension in ϕ according to the defined ordering. For example, for $\phi = B\langle A\langle \texttt{Int}, \texttt{Bool}\rangle, \texttt{Bool}\rangle$, we have $A{\downarrow}\phi$. With this constraint, we can write the type for *bounded* as follows.

$$\textit{bounded} : \forall d.d{\downarrow}\phi \Rightarrow \phi \to d\langle \texttt{Int}, \texttt{Bool}\rangle$$

This type states that for any given argument, there is at most one valid instantiation of d. Note that the constraint $d{\downarrow}\phi$ must be satisfied only if d appears in the result type, as it does in *bounded*. In general, the conditional constraint $\delta \in \phi_2 \Rightarrow \delta{\downarrow}\phi_1$ means that $\delta{\downarrow}\phi_1$ must be satisfied only if $\delta \in \phi_2$ is satisfied.

We define other constraints in Figure 6 and use the metavariable C to range over constraints. The constraint $\phi_1 \equiv \phi_2$ requires that types ϕ_1 and ϕ_2 be equivalent, which is more flexible than type equality. The "variational constraint" $\delta\langle C_1, C_2\rangle$ expresses a choice between constraint C_1 or C_2, depending on dimension δ. The constraint forms $C_1 \wedge C_2$ and $\exists\overline{\alpha}\overline{d}.C$ have the conventional meanings.

[3] E.g., we don't deal with $A\langle \forall\alpha.\alpha \to \alpha, \forall\alpha.\alpha \to \texttt{Int}\rangle$ since we can transform it to $\forall\alpha,\beta.A\langle\alpha \to \alpha, \beta \to \texttt{Int}\rangle$.

$$\text{E1}\quad C \Vdash C \qquad \text{E2}\quad C \wedge C_1 \Vdash C_1 \qquad \text{E3}\quad C \Vdash \exists\overline{\alpha}\overline{d}.C \qquad \text{E4}\ \frac{C_1 \Vdash C_2 \qquad C_2 \Vdash C_3}{C_1 \Vdash C_3} \qquad \text{E5}\ \frac{C_1 \Vdash C_2}{\theta(C_1) \Vdash \theta(C_2)}$$

$$\text{E6}\ \frac{C_1 \Vdash C_2}{\exists\overline{\alpha}\overline{d}.C_1 \Vdash \exists\overline{\alpha}\overline{d}.C_2} \qquad \text{E7}\ \frac{C_1 \Vdash C_2 \qquad C_3 \Vdash C_4}{\delta\langle C_1, C_3\rangle \Vdash \delta\langle C_2, C_4\rangle}$$

$$\text{E8}\ \frac{FD(\phi_1) = \varnothing \qquad FV(\phi_1) = \varnothing \qquad \delta \notin dims(\phi_1)}{\Vdash \delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi} \qquad \text{E9}\ \frac{\delta \in dims(\phi_1) \qquad \delta = \min(dims(\phi))}{\Vdash \delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi}$$

$$\text{T1}\quad \phi_1 \equiv \phi_2 \Vdash \phi_2 \equiv \phi_1 \qquad \text{T2}\quad \phi_1 \equiv \phi_2 \wedge \phi_2 \equiv \phi_3 \Vdash \phi_1 \equiv \phi_3 \qquad \text{T3}\quad \Vdash \phi \equiv \phi$$

$$\text{T4}\quad \phi_1 \equiv \phi_2 \Vdash \phi[\phi_1] \equiv \phi[\phi_2] \qquad \text{T5}\quad \Vdash \delta\langle\phi, \phi\rangle \equiv \phi \qquad \text{T6}\quad \Vdash \delta\langle\phi_1, \phi_2\rangle \equiv \delta\langle\lfloor\phi_1\rfloor_{\delta_\ell}, \lfloor\phi_2\rfloor_{\delta_r}\rangle$$

Figure 7 Entailment relation of constraints

We now turn to the relations among constraints. We use the entailment relation $C_1 \Vdash C_2$ to denote that if C_1 is satisfied, then C_2 must also be satisfied. The rules E1, E2, and E4 are standard in constrained type systems. The rules E3 and E6 specify that existential quantifiers hide information of constraints and thus quantified constraints are easier to satisfy. The constraint E5 states that the entailment relation is stable under substitution, where θ is a mapping of type variables to types and dimension variables to dimensions. Rule E7 deals with variational constraints and is obvious.

Finally, rules E8 and E9 describe two ways to satisfy the constraint $\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi$. The first applies when δ doesn't occur in ϕ_1, which we can conclude only when ϕ_1 doesn't contain dimension variables or type variables because they may be substituted with other variational types. The operation $FD(\phi)$ is defined as follows.

$$\begin{aligned}
FD(d\langle\phi_1, \phi_2\rangle) &= \{d\} \cup FD(\phi_1) \cup FD(\phi_2)\\
FD(D\langle\phi_1, \phi_2\rangle) &= FD(\phi_1) \cup FD(\phi_2)\\
FD(\phi_1 \to \phi_2) &= FD(\phi_1) \cup FD(\phi_2)\\
FD(d) &= \{d\} \qquad FD(D) = FD(\tau) = \varnothing
\end{aligned}$$

The operation $dims(\phi)$ is defined similarly but collecting all δs. The second situation is handled by E9, which requires that $\delta \in dims(\phi_1)$ and that δ is $\min(dims(\phi_2))$.

The second part of Figure 7 presents the entailment relation between type equivalence relations. The rules T1 through T3 state that type equivalence is reflexive, symmetric, and transitive. Rule T4 describes that equivalence is congruent with respect to an arbitrary shared context. The rules T5 and T6 state that idempotent choices and dead alternatives may be eliminated. The operation $\lfloor\phi_1\rfloor_{\delta_\ell}$ replaces each occurrence of a choice in dimension δ within ϕ_1 with its left alternative [17].

4.2 Typing Rules

We use the judgment $C; \Gamma; \Delta \vdash e : \phi$ to denote that under the constraint C, the type environment Γ, and the dimension environment Δ, the expression e has the type ϕ. The environment

$$\boxed{C;\Gamma;\Delta \vdash e : \phi}$$

$$\textsc{Con}\ \frac{\kappa \ \text{ of type } \gamma}{C;\Gamma;\Delta \vdash \kappa : \gamma} \qquad \textsc{Var}\ \frac{\Gamma(v) = \forall\overline{\alpha}\overline{d}.C \Rightarrow \phi \qquad C' \Vdash C}{C';\Gamma;\Delta \vdash v : \phi} \qquad \textsc{Abs}\ \frac{C;\Gamma,(v,\phi_1);\Delta \vdash e : \phi}{C;\Gamma;\Delta \vdash \lambda v.e : \phi_1 \to \phi}$$

$$\textsc{App}\ \frac{C;\Gamma;\Delta \vdash e_1 : \phi_1 \qquad C;\Gamma;\Delta \vdash e_2 : \phi_2 \qquad C \Vdash \phi_1 \equiv \phi_2 \to \phi}{C;\Gamma;\Delta \vdash e_1\ e_2 : \phi} \qquad \textsc{AbsD}\ \frac{C;\Gamma;\Delta,(d,\delta) \vdash e : \phi}{C;\Gamma;\Delta \vdash \lambda d.e : \delta \to \phi}$$

$$\textsc{Let}\ \frac{\overline{d}\ \#\ FD(C_2) \cup FD(\Gamma) \qquad \overline{\alpha}\ \#\ FV(C_2) \cup FV(\Gamma) \qquad \begin{array}{c} C_1 \wedge C_2;\Gamma,(v,\phi);\Delta \vdash e_1 : \phi \\ C_2;\Gamma,(v,\forall\overline{\alpha}\overline{d}.C_1 \Rightarrow \phi);\Delta \vdash e_2 : \phi_2 \end{array}}{C_2 \wedge \exists\overline{\alpha}\overline{d}.C_1;\Gamma;\Delta \vdash \textbf{let}\ v = e_1\ \textbf{in}\ e_2 : \phi_2}$$

$$\textsc{Dim}\ \frac{\Delta(\delta) = \delta_1}{C;\Gamma;\Delta \vdash \delta : \delta_1} \qquad \textsc{Chc}\ \frac{C_1;\Gamma;\Delta \vdash e_1 : \phi_1 \qquad C_2;\Gamma;\Delta \vdash e_2 : \phi_2 \qquad \Delta(\delta_1) = \delta}{\delta\langle C_1, C_2\rangle;\Gamma;\Delta \vdash \delta_1\langle e_1, e_2\rangle : \delta\langle \phi_1, \phi_2\rangle}$$

$$\textsc{Sel}\ \frac{C;\Gamma;\Delta \vdash e : \phi \qquad \Delta(\delta) = \delta_1 \qquad C \Vdash \delta_1\langle \phi_\ell, \phi_r\rangle \equiv \phi \qquad \begin{cases} C \Vdash \alpha \equiv \phi_\ell & s = \ell \\ C \Vdash \alpha \equiv \phi_r & s = r \end{cases}}{C;\Gamma;\Delta \vdash \delta_s \blacktriangleright e : \alpha}$$

$$\textsc{Any}\ \frac{C;\Gamma;\Delta,(d,d) \vdash e_2 : \phi_2 \qquad \begin{array}{c} C;\Gamma;\Delta \vdash e_1 : \phi_1 \\ C;\Gamma;\Delta \vdash e_3 : \phi_3 \end{array} \qquad C \Vdash \phi_2 \equiv \phi_3 \qquad C \Vdash d \in \phi_3 \Rightarrow d{\downarrow}\phi_1}{C;\Gamma;\Delta \vdash \textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 : \phi_3}$$

$$\textsc{The}\ \frac{C;\Gamma;\Delta \vdash e_1 : \phi_1 \qquad \Delta(\delta) = \delta_1 \qquad C;\Gamma;\Delta \vdash e_2 : \phi_2 \qquad C;\Gamma;\Delta \vdash e_3 : \phi_3 \qquad C \Vdash \phi_2 \equiv \phi_3}{C;\Gamma;\Delta \vdash \textbf{the}\ \delta\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 : \phi_3}$$

Figure 8 Typing rules

Δ that collects all the assumptions for bound dimension variables that are visible for e is a mapping from dimension variables d to dimensions δ. We frequently use the operation $\Delta(\delta)$, which is defined as follows.

$$\Delta(\delta) = \begin{cases} D & \text{if } \delta = D \\ \delta_1 & \text{if } \delta = d \wedge (d,\delta_1) \in \Delta \\ \text{undefined} & \text{if } \delta = d \wedge d \notin \mathit{dom}(\Delta) \end{cases}$$

The first three rules (Con, Var, and Abs) are standard. The rule App for function application relaxes the usual equality constraint between the function parameter and argument and requires only that they be equivalent (not equal). The rule AbsD is similar to Abs except that it abstracts over dimension variables. To type dimension abstractions, we assume a binding for (d,δ) in Δ, then type the body.

The Let rule deals with recursive **let** expressions. Note that we allow polymorphism over both dimension variables and type variables. The overall typing process is standard. However, note that we only attach the constraint (C_1) that specifies requirements for $\overline{\alpha}$ and $\overline{d}$ to the type ϕ assumed for v. Since the variable v will probably be referred to many times in the body, this helps to control the size of the constraint aggregated during the typing process. As been noted earlier [40, 44, 13], the size of constraints has a huge impact on the performance of type checkers. We therefore regard this optimization as necessary. Another subtlety is that we keep the constraint $\exists\overline{\alpha}\overline{d}.C_1$ in the result typing constraint. This constraint increases the precision of the type system. For a discussion, see [40, 44].

Rule Dim consults the binding for δ in Δ. From the definition of $\Delta(\delta)$ it follows that $C;\Gamma;\Delta \vdash D : D$ and $C;\Gamma;\Delta \vdash d : D$ if $(d, D) \in \Delta$. The rule Chc for choice construction expresses the idea that the typing process can be composed over choice creation. Specifically, if e_1 and e_2 are typed under the constraints C_1 and C_2, respectively, then $\delta_1\langle e_1, e_2\rangle$ is well typed under the variational constraint $\delta\langle C_1, C_2\rangle$, where $\delta = \Delta(\delta_1)$. To type a selection, we first retrieve the type ϕ_1 of the expression being selected. Then, based on the selector given, the constraint must ensure that the result type is equivalent to the corresponding alternative of ϕ_1. This idea is formalized in rule Sel.

To type an **any** expression, we first derive the type ϕ_1 for e_1. Then we have to consider two different cases. First, if ϕ_1 contains a dimension, we type the **then** branch e_2 by extending Δ with (d, d). Otherwise, if ϕ_1 doesn't contain any dimension, then the **else** branch e_3 is typed. The language requires that both branches have the same type. Another constraint that needs to be satisfied is that if the result type of the expression refers to d, then d must appear in ϕ_1 and must be the smallest dimension according to the ordering of dimensions defined in Section 3. This is expressed through the entailment relation $C \Vdash d \in \phi_3 \Rightarrow d{\downarrow}\phi_1$. The rule Any specifies this typing process.

Note that the result will be incorrect if we drop the condition and use the constraint entailment $C \Vdash d{\downarrow}\phi_1$. To illustrate, consider the expression `vmin` introduced in Section 2.4. Although we will not describe the formal reasoning process, we briefly discuss how the type for `vmin` can be derived. First, since `vmin` is a recursive function, we assume its type is $\alpha \to \beta$. Next, the call of `min` of result type β forces β to be `Int` and the return type of the **then** branch to be `Int`. As both branches of **any** need to have the same type, the **else** branch of type α also has the type `Int`. Thus, both α and β are `Int`. If the **any** expression introduces the constraint $d{\downarrow}\texttt{Int}$, we observe that the constraint can never be satisfied since `Int` doesn't contain any dimension. However, if we introduce the constraint $d \in \texttt{Int} \Rightarrow d{\downarrow}\texttt{Int}$, then the constraint can immediately be removed according to rule E8 in Figure 7.

The idea of typing **the** expressions is very similar to that of typing **any** expressions. Both branches of **the** also need to be equivalent. The main difference is that this rule doesn't place a constraint on the dimension δ, as can be seen from the rule The. The reason is that δ must be bound by a dimension abstraction. Note that while other judgments in The don't use δ_1, we write $\Delta(\delta) = \delta_1$ to ensure that δ is a dimension constant or is bound in Δ.

Our type system is sound with respect to the operational semantics defined in Section 3.2. Specifically, the following two properties hold.

▸ **Theorem 2** (Progress). *If $C;\Gamma;\varnothing \vdash e : \phi$ and Γ contains information for constants only and $FV(\Gamma) = \varnothing$, and $\Vdash C$, then e is a value or there is some e' such that $e \longrightarrow e'$.*

▸ **Theorem 3** (Preservation). *If $C;\Gamma;\Delta \vdash e : \phi$ and $e \longrightarrow e'$, then $C;\Gamma;\Delta \vdash e' : \phi$.*

Progress can be proved by induction on the typing derivation, and preservation by induction on the reduction relation.

$$\textsc{I-App}\ \frac{C_1;\Gamma;\Delta \vdash_I e_1 : \alpha_1 \qquad C_2;\Gamma;\Delta \vdash_I e_2 : \alpha_2 \qquad \alpha_3 \text{ fresh}}{\exists \alpha_1\alpha_2.(C_1 \wedge C_2 \wedge \alpha_1 \equiv \alpha_2 \to \alpha_3);\Gamma;\Delta \vdash_I e_1\ e_2 : \alpha_3}$$

$$\textsc{I-Any}\ \frac{\begin{array}{c} C_1;\Gamma;\Delta \vdash_I e_1 : \alpha_1 \\ C_2;\Gamma;\Delta,(d,d) \vdash_I e_2 : \alpha_2 \qquad C_3;\Gamma;\Delta \vdash e_3 : \alpha_3 \qquad C_4 = (d \in \alpha_3 \Rightarrow d{\downarrow}\alpha_1) \end{array}}{\exists \alpha_1\alpha_2 d.(C_1 \wedge C_2 \wedge C_3 \wedge \alpha_2 \equiv \alpha_3 \wedge C_4);\Gamma;\Delta \vdash_I \textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 : \alpha_3}$$

Figure 9 Constraint generation for VPC

5 Constraint Generation

The type inference process consists of two steps: constraint generation and constraint solving. This section presents constraint generation and investigates its properties. Constraint solving will be presented in Section 6. The goal of constraint generation is to collect constraints for a specific expression e under the type environment Γ and dimension environment Δ. Figure 9 presents constraint generation rules for function applications and **any** expressions. The full set of constraint generation rules is presented in Appendix A.3. The rules define the judgment $C;\Gamma;\Delta \vdash_I e : \alpha$, which means that given e, Γ, and Δ, the constraint C will be collected and the result type is α.

The constraint generation rules are derived from the typing rules in Figure 8. The constraints for a given expression e are a combination of the constraints of its subexpressions and constraints relating the types of its subexpressions. For example, in the I-App rule, the constraints for $e_1\ e_2$ are obtained as the conjunction of the constraints C_1, C_2, and $\alpha_1 \equiv \alpha_2 \to \alpha_3$, where C_1 and C_2 are the constraints for e_1 and e_2, respectively, and the constraint $\alpha_1 \equiv \alpha_2 \to \alpha_3$ relates the types of e_1, e_2, and $e_1\ e_2$.

An important observation is that the constraint generation rules collect exactly the constraints specified in the typing relation in Figure 8—they do not forget or introduce new constraints. This leads to the following soundness and completeness theorems for constraint generation.

▸ **Theorem 4** (Soundness of constraint generation). *If $C;\Gamma;\Delta \vdash_I e : \alpha$, then $C;\Gamma;\Delta \vdash e : \alpha$.*

▸ **Theorem 5** (Completeness and principality of constraint generation). *If $C;\Gamma;\Delta \vdash e : \phi$, then $C';\Gamma;\Delta \vdash_I e : \alpha$ with $C \Vdash \exists \alpha.C'$ and $C \wedge C' \Vdash \theta(\alpha) \equiv \phi$ for some substitution θ.*

Theorem 4 can be proved by induction over the constraint generation rules, Theorem 5 by induction over the typing rules.

Based on the constraint generation rules, the following set of constraints (reformatted for readability) will be generated for the expression *bounded* $A\langle 2,3\rangle$ (where *bounded* is the function introduced in Section 4.1). We refer to these constraints collectively as $\mathcal{C}_1$ (and use $\mathcal{C}$ to range over constraint sets henceforth). Here α and α_1 represent the return type and argument type of *bounded*, respectively.

$$C_1 : \alpha \equiv d\langle \texttt{Int},\texttt{Bool}\rangle \qquad C_2 : d \in d\langle \texttt{Int},\texttt{Bool}\rangle \Rightarrow d{\downarrow}\alpha_1 \qquad C_3 : \alpha_1 \equiv \alpha_2 \qquad C_4 : \alpha_2 \equiv A\langle \texttt{Int},\texttt{Int}\rangle$$

The constraint set $\mathcal{C}_2$ for the application *bounded* 2 has the same first three constraints as $\mathcal{C}_1$ and has a different last constraint C_5.

$$C_1 : \alpha \equiv d\langle \texttt{Int},\texttt{Bool}\rangle \qquad C_2 : d \in d\langle \texttt{Int},\texttt{Bool}\rangle \Rightarrow d{\downarrow}\alpha_1 \qquad C_3 : \alpha_1 \equiv \alpha_2 \qquad C_5 : \alpha_2 \equiv \texttt{Int}$$

6 Constraint Solving

While generating constraints from the typing is straightforward, solving such constraints is more challenging. Section 6.1 discusses these challenges, and Section 6.2 presents a constraint solver.

6.1 Constraint Solving Challenges

There are three main challenges to address. First, rule T5 in Figure 7 specifies that we can always simplify a type by eliminating idempotent choices. Moreover, doing so can significantly improve the performance of type inference algorithms in practice, as has been shown in [17]. However, eagerly eliminating all idempotent choices can render some satisfiable constraints unsatisfiable. Consider, for example, the constraint set $\mathcal{C}_1$ generated in Section 5. If we solve C_3 and C_4 with the substitution $\theta = \{\alpha_2 \mapsto \texttt{Int}, \alpha_1 \mapsto \texttt{Int}\}$, then C_2 cannot be satisfied since $d{\downarrow}\texttt{Int}$ fails because `Int` doesn't contain any dimension. This is undesirable since the expression that generates $\mathcal{C}_1$ is well typed. However, if we instead use the substitution $\theta' = \{\alpha_2 \mapsto A\langle\texttt{Int},\texttt{Int}\rangle, \alpha_1 \mapsto A\langle\texttt{Int},\texttt{Int}\rangle\}$, then C_2 can be solved by mapping d to A. Finally, solving C_1 gives us $A\langle\texttt{Int},\texttt{Bool}\rangle$ as the result type of *bounded* $A\langle 2, 3\rangle$. This result matches our expectation since the expression is indeed well typed.

On the other hand, we shouldn't introduce gratuitous idempotent choices, because those might allow solving some unsatisfiable constraints. Consider, for example, the constraint set $\mathcal{C}_2$ from Section 5. We again have both θ and θ' as potential solutions for $\mathcal{C}_2$. Since the expression that generates $\mathcal{C}_2$ is ill typed, only θ leads to the expected result since it makes C_2 unsolvable.

The second challenge regards the kind of information produced through constraint solving. This is straightforward for constraints on type variables. For example, in the case of $A\langle\texttt{Int},\alpha\rangle \equiv A\langle\alpha,\texttt{Bool}\rangle$, we simply find a substitution for α. But what do we do about $d_1\langle\texttt{Int},\texttt{Bool}\rangle \equiv d_2\langle\texttt{Int},\texttt{Bool}\rangle$? At best, this constraint allows us to derive that $d_1 = d_2$. However, in general such constraints are undecidable. For example, consider $A\langle\texttt{Int},\alpha\rangle \equiv d\langle\texttt{Int},\beta\rangle$. Both $\{d \mapsto A, \alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}\}$ and $\{d \mapsto B, \alpha \mapsto \texttt{Int}, \beta \mapsto \texttt{Int}\}$ are possible substitutions but neither is more general than the other.

The third challenge is that, unlike in normal constraint solving [40], a set of constraints cannot be solved in one iteration. Consider, for example, the following constraint set $\mathcal{C}_3$.

$$C_6 : d_1\langle\texttt{Int},\texttt{Bool}\rangle \equiv d_2\langle\texttt{Int},\texttt{Bool}\rangle \qquad\qquad C_7 : d_1 \equiv A \wedge d_2 \equiv B$$

For the same reason as in the previous paragraph*, solving C_6 directly won't provide useful information. We can solve C_7, however, with $d_1 \mapsto A$ and $d_2 \mapsto B$. By substituting d_1 with A and d_2 with B in C_6, we can now find that C_6 is unsolvable. Overall, we need two iterations to determine that $\mathcal{C}_3$ is unsolvable.

To address the first challenge, we require that idempotent choices are not reduced when solving certain constraints. To address the second challenge and make constraint solving decidable, we defer computing solutions for dimension variables when they are used as choice constructors. Finally, to address the third challenge, we use iterative constraint solving, that is we solve constraints in multiple iterations and stop when no further progress can be made.

To simplify the presentation, in addition to the constraint forms in Figure 6, we introduce a new constraint form, $d{\downarrow}\phi$, which is satisfied if $d = \min(\mathit{dims}(\phi))$. We also use `true` to denote a constraint that is trivially satisfied. We define the auxiliary function $\mathit{noelim}(\mathcal{C},\mathcal{K})$ in Figure 10 to decide when idempotent choices shouldn't be removed. Given $\mathcal{C}$, the iterative function $\mathit{noelim}(\mathcal{C},\{\})$ returns the set of type variables for which we must preserve idempotent choices.

$$
\begin{aligned}
\mathit{noelim}(\phi_1 \equiv \phi_2, \mathcal{K}) &= \begin{cases} \mathcal{K} & \mathcal{K} \cap FV(\phi_1, \phi_2) = \varnothing \\ \mathcal{K} \cup FV(\phi_1, \phi_2) & \text{otherwise} \end{cases} \\
\mathit{noelim}(\exists \overline{\alpha}\overline{d}.C, \mathcal{K}) &= \mathit{noelim}(C, \mathcal{K} - \overline{\alpha}) \\
\mathit{noelim}(C_1 \wedge C_2, \mathcal{K}) &= \mathit{noelim}(\delta\langle C_1, C_2\rangle, \mathcal{K}) = \mathit{noelim}(C_1, \mathcal{K}) \cup \mathit{noelim}(C_2, \mathcal{K}) \\
\mathit{noelim}(\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi, \mathcal{K}) &= \mathit{noelim}(\delta{\downarrow}\phi, \mathcal{K}) = \mathcal{K} \cup FV(\phi) \\
\mathit{noelim}(\mathcal{C}, \mathcal{K}) &= \begin{cases} \mathcal{K} & \mathcal{K} = \mathcal{K}' \\ \mathit{noelim}(\mathcal{C}, \mathcal{K}') & \text{otherwise} \end{cases} \qquad \text{where } \mathcal{K}' = \bigcup_{C \in \mathcal{C}} \mathit{noelim}(C, \mathcal{K})
\end{aligned}
$$

Figure 10 Keeping idempotent choices

Intuitively, $\mathit{noelim}(\mathcal{C}, \{\})$ includes (1) the type variables found in variational types ϕ for constraints of the form $\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi$ or $\delta{\downarrow}\phi$ and (2) the type variables that share a constraint with some other type variables in $\mathit{noelim}(\mathcal{C}, \{\})$. $\mathit{noelim}(\mathcal{C}, \{\})$ is complete since it handles all different forms of the constraint.

Given this operation, we can compute $\mathit{noelim}(\mathcal{C}_1, \{\}) = \{\alpha_1, \alpha_2\}$, where $\mathcal{C}_1$ is from Section 5. Note that α is not included in the resulting set since it shares no constraint with either α_1 or α_2.

6.2 A Constraint Solver

We can now implement a constraint solver $\mathcal{S}$. Given a constraint set $\mathcal{C}$ and a mapping θ, $\mathcal{S}$ solves iteratively until no progress can be made, which is detected when $\mathcal{C} = \mathcal{C}'$ where $\mathcal{C}'$ is the residual constraint set of the current iteration. Note that a new $\mathit{noelim}(\mathcal{C}, \{\})$ is computed and passed to $\mathcal{U}'$ at the beginning of each iteration since constraints may have been changed during the previous iteration.

$$
\mathcal{S}(\mathcal{C}, \theta) = \begin{cases} (\mathcal{C}', \theta') & \mathcal{C} = \mathcal{C}' \\ \mathcal{S}(\mathcal{C}', \theta') & \text{otherwise} \end{cases} \qquad \text{where } (\mathcal{C}', \theta') = \mathcal{U}'(\mathcal{C}, \theta, \mathit{noelim}(\mathcal{C}, \{\}))
$$

The real work of solving constraints is performed by $\mathcal{U}'$, which in turn calls $\mathcal{U}$ to solve each constraint individually. Both $\mathcal{U}'$ and the main part of $\mathcal{U}$ are given in Figure 11. Several simple cases, such as unifying two plain types and unifying two dimension variables, for $\mathcal{U}$ are omitted. Given $\mathcal{C}$, $\mathcal{U}'$ will return the same result if the constraints are solved in different orderings. However, if $\mathcal{C}$ fails to solve, $\mathcal{U}'$ will fail on different constraints considering different orderings. The main solver $\mathcal{U}$ has the type $\mathcal{U} : C \times \theta \times 2^{\alpha} \to C \times \theta$, that is, it takes three arguments—the constraint to be solved, the mapping, and the set of type variables for handling idempotent choice eliminations—and returns as two results when constraint solving is successful the residual constraint and the result unifier.

We go briefly over the cases in Figure 11. Case (a) deals with existential constraints of the form $\exists \alpha d.C$, which is satisfiable if C is satisfiable. For this constraint, we first solve C and remove binding information about α and d in the result unifier. Case (b) states that a conditional constraint $\delta \in \phi_1 \Rightarrow \delta{\downarrow}\phi_2$ can be simplified in two ways: (1) if the condition $\delta \in \phi_1$ is satisfied, the constraint is simplified to $\delta{\downarrow}\phi_2$ and (2) if the condition fails, the whole constraint is satisfied. Otherwise, the constraint is deferred for later iterations if the condition contains free dimension or type variables, which may be substituted with concrete dimension names or types.

$$\begin{aligned}\mathcal{U}'(\{C_1,\dots,C_n\},\theta,\mathcal{K}) = \textbf{let } & (C'_1,\theta'_1) = \mathcal{U}(C_1,\theta,\mathcal{K})\\ & (\mathcal{C}',\theta') = \mathcal{U}'(\theta'_1(\{C_2,\dots,C_n\}),\theta'_1,\mathcal{K})\\ \textbf{in } & (\{C'_1\}\cup\mathcal{C}',\theta')\end{aligned}$$

(a) $\mathcal{U}(\exists\alpha d.C,\theta,\mathcal{K}) = \textbf{let } (C_1,\theta_1) = \mathcal{U}(C,\theta,\mathcal{K}) \textbf{ in } (\exists\alpha d.C_1,\theta_1\backslash\{\alpha,d\})$

(b) $\mathcal{U}(\delta\in\phi_1 \Rightarrow \delta{\downarrow}\phi_2,\theta,\mathcal{K})$
$\quad\mid \delta\in \mathit{dims}(\theta(\phi_1)) = \mathcal{U}(\delta{\downarrow}\phi_2,\theta,\mathcal{K})$
$\quad\mid \mathit{FD}(\theta(\phi_1)) == \varnothing \land \mathit{FV}(\theta(\phi_1)) == \varnothing \land \delta\notin \mathit{dims}(\theta(\phi_1)) = (\mathit{true},\theta)$
$\quad\mid \textbf{otherwise } = (\delta\in\phi_1 \Rightarrow \delta{\downarrow}\phi_2,\theta)$

(c) $\mathcal{U}(\delta{\downarrow}\phi_2,\theta,\mathcal{K})$
$\quad\mid \mathit{FD}(\theta(\phi_2)) \neq \varnothing \lor \mathit{FV}(\theta(\phi_2)) \neq \varnothing = (\delta{\downarrow}\phi_2,\theta)$
$\quad\mid \mathit{dims}(\theta(\phi_2)) == \varnothing = \text{fail}$
$\quad\mid \delta == d = (\mathit{true},\{d\mapsto \min(\mathit{dims}(\theta(\phi_2)))\}\circ\theta)$
$\quad\mid \delta \neq \min(\mathit{dims}(\theta(\phi_2))) = \text{fail}$
$\quad\mid \textbf{otherwise } = (\mathit{true},\theta)$

(d) $\mathcal{U}(D_1\langle\phi_1,\phi_2\rangle \equiv D_2\langle\phi_3,\phi_4\rangle,\theta,\mathcal{K}) =$
$\quad\textbf{let } (\theta_1,r) = \mathit{vunify}(D_1\langle\phi_1,\phi_2\rangle,D_2\langle\phi_3,\phi_4\rangle) \textbf{ in if } r \textbf{ then } (\mathit{true},\theta_1\circ\theta) \textbf{ else } \text{fail}$

(e) $\mathcal{U}(\alpha\equiv\phi,\theta,\mathcal{K}) = \textbf{if } \alpha\in\mathcal{K} \textbf{ then } (\mathit{true},\{\alpha\mapsto\phi\}\circ\theta) \textbf{ else } (\mathit{true},\{\alpha\mapsto \mathit{norm}(\phi)\}\circ\theta)$

(f) $\mathcal{U}(\delta\langle C_1,C_2\rangle,\theta,\mathcal{K}) =$
$\quad\textbf{let } (C_3,\theta_3) = \mathcal{U}(C_1,\theta,\mathcal{K})\ ;\ (C_4,\theta_4) = \mathcal{U}(C_2,\theta,\mathcal{K})\ \textbf{ in } (\delta\langle C_3,C_4\rangle,\delta\langle\theta_3,\theta_4\rangle)$

(g) $\mathcal{U}(C_1\land C_2,\theta,\mathcal{K}) =$
$\quad\textbf{let } (C_3,\theta_3) = \mathcal{U}(C_1,\theta,\mathcal{K})\ ;\ (C_4,\theta_4) = \mathcal{U}(\theta_3(C_2),\theta_3,\mathcal{K}) \textbf{ in } (C_3\land C_4,\theta_4\circ\theta_3)$

(h) $\mathcal{U}(C,\theta,\mathcal{K}) = (C,\theta)$

Figure 11 A constraint solving algorithm using pattern matching and guard expressions.

Case (c), for solving constraints of the form $\delta{\downarrow}\phi_2$, is more interesting. This constraint is processed only when $\theta(\phi_2)$ doesn't contain any free dimensions or type variables (handled by the first guard and its body). Otherwise, if the substituted type doesn't contain any dimensions, then solving fails (handled by the second guard and its body). For example, constraint solving for $\delta{\downarrow}$`Int` will fail since `Int` doesn't contain any dimension. Next, if δ is a dimension variable d, then d is mapped to the minimum dimension and the result unifier is updated (handled by the third guard and its body). For example, the solution for $\delta{\downarrow}A\langle$`Int`,`Bool`$\rangle$ is $\delta\mapsto A$. Otherwise, if δ doesn't match the minimum dimension, then the constraint is unsatisfiable (handled by the fourth guard and its body). For example, $B{\downarrow}A\langle$`Int`,`Bool`$\rangle$ is unsolvable. Finally, if δ is the same as the minimum dimension of ϕ_2, the constraint is trivially satisfiable (last case).

In contrast to previous situations, δ is the same as the minimum dimension, which is handled by the last guard and the corresponding body.

Case (d) deals with type equivalence when both types are variational and contain dimension literals only. In this case, we delegate the work to *vunify*, the variational unification algorithm from [14]. Given two types ϕ_1 and ϕ_2, $\mathit{vunify}(\phi_1,\phi_2)$ returns a pair (θ,r), where θ is the unifier when the unification is successful and r is a boolean value indicating whether the unification problem is solved successfully. The use of *vunify*, together with the fact that type equivalence constraints containing dimension variables are deferred until they are instantiated with dimension literals, makes the constraint solver here simpler than the unification algorithm in VLC [17]. A detailed discussion of the relationship between the constraint solver and VLC can be found in Appendix A.4.

Case (e) solves a type equivalence constraint between a type variable α and a variational type ϕ, where α doesn't occur in ϕ. In this case, we further decide if $\alpha \in \mathcal{K}$. If so, α is mapped to ϕ. Otherwise, ϕ is simplified with the auxiliary function $norm(\phi)$, and α is mapped to the simplified type. The auxiliary function $norm(\phi)$ reduces idempotent choices and dead alternatives, which are discussed in more detail in [17]. For example, $\mathcal{U}(\alpha \equiv A\langle \texttt{Int}, \texttt{Int}\rangle, \varnothing, \{B\})$ yields $(\texttt{true}, \{\alpha \mapsto \texttt{Int}\})$, and $\mathcal{U}(\alpha \equiv B\langle \texttt{Int}, \texttt{Int}\rangle, \varnothing, \{B\})$ yields $(\texttt{true}, \{\alpha \mapsto B\langle \texttt{Int}, \texttt{Int}\rangle\})$.

Both cases (f) and (g) are straightforward. To solve a variational constraint, we just solve each alternative constraint. To solve the constraint $C_1 \wedge C_2$, we solve both C_1 and C_2. Although the structures for these cases are very similar, there exist two subtle differences. First, C_1 and C_2 are solved independently in (f) while they are solved sequentially in (g). Second, (f) returns a variation of the substitutions for C_1 and C_2 while (g) returns a composition of the substitutions. These differences reflect that the constraints in a variational constraint are independent of each other. Other than previous cases, the constraint is deferred to later iterations, as can be seen in case (h).

$\mathcal{U}$ is sound, complete, and principal, captured in the following theorems.

▸ **Theorem 6** (Soundness of $\mathcal{U}$)**.** *If $(C_2, \theta_2) = \mathcal{U}(C_1, \theta_1, \mathcal{K})$, then $C_2 \Vdash \theta_2(C_1)$ and $\theta_2(C_2) = C_2$.*

▸ **Theorem 7** (Completeness and principality of $\mathcal{U}$)**.** *Given (C_1, θ_1), if $C_3 \Vdash \theta_3(C_1)$, then $(C_2, \theta_2) = \mathcal{U}(C_1, \theta_1, \mathcal{K})$ and $\theta_2 \sqsubseteq \theta_3$ and $C_3 \Vdash \theta_3(C_2)$, where $\theta_2 \sqsubseteq \theta_3$ if there is some θ_4 such that $\theta_3 = \theta_4 \circ \theta_2$.*

Both theorems can be proved by induction on the constraint solving rules in Figure 11.

With $\mathcal{U}'$ and $\mathcal{U}$, we can illustrate how $\mathcal{S}$ solves $\mathfrak{C}_1$ from Section 5 through the following sequence of transformations.

$$\begin{gathered}
(\mathfrak{C}_1, \varnothing, \mathcal{K}) \xrightarrow{(e),\textbf{else}} (\{C_2, C_3, C_4\}, \{\alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \mathcal{K}) \xrightarrow{(b),1} \\
(\{C_3, C_4; d{\downarrow}\alpha_1\}, \{\alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \mathcal{K}) \xrightarrow{\qquad} \\
(\{C_4; d{\downarrow}\alpha_1\}, \{\alpha_1 \mapsto \alpha_2, \alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \mathcal{K}) \xrightarrow{(e),\textbf{then}} \\
(\{d{\downarrow}\alpha_1\}, \{\alpha_2 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha_1 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha \mapsto d\langle \texttt{Int}, \texttt{Bool}\rangle\}, \{\alpha_1\}) \xrightarrow{(c),3} (\texttt{true}, \theta_4, \{\alpha_1\}) \\
\theta_4 = \{\alpha_1 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, \alpha_2 \mapsto A\langle \texttt{Int}, \texttt{Int}\rangle, d \mapsto A, \alpha \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle\}
\end{gathered}$$

The constraint solving process begins with $(\mathfrak{C}_1, \varnothing, \mathcal{K})$, where $\mathcal{K} = noelim(\mathfrak{C}_1, \{\}) = \{\alpha_1, \alpha_2\}$. Constraint C_1 from $\mathfrak{C}_1$ is solved by the **else** branch of case (e) in Figure 11, as indicated by the label "(e),**else**" over the arrow. Next C_2 is solved by the first case of (b), which produces the residual constraint $d{\downarrow}\alpha_1$ that is included in the constraint set. It is separated by a ";" to indicate that the constraint will be solved in the second iteration. Since the case of unifying two type variables is not included in Figure 11, the arrow for solving C_3 is not labelled. The constraint solving process ends with $(\texttt{true}, \theta_4, \{\alpha_1\})$, where θ_4 is shown at the end of the transformation. This result tells us that the expression *bounded* $A\langle \texttt{2}, \texttt{3}\rangle$ is well typed and has the type $\theta_4(\alpha) = A\langle \texttt{Int}, \texttt{Bool}\rangle$.

On the other hand, $\mathcal{S}(\mathfrak{C}_2, \{\})$ fails because the constraint solving process leads to the constraint $d{\downarrow}\texttt{Int}$, which is unsolvable according to the second case of rule (c). Similarly, $\mathcal{S}(\mathfrak{C}_3, \{\})$ fails in the second iteration since the constraint $A\langle \texttt{Int}, \texttt{Bool}\rangle \equiv B\langle \texttt{Int}, \texttt{Bool}\rangle$ is unsatisfiable.

In general, $\mathcal{S}$ is sound, complete, and principal since it inherits these properties from $\mathcal{U}$. Together with Theorems 4 and 5, it follows that type inference is sound, complete, and principal.

7 Related Work

In Section 1, we provided references to many applications of variational programming. However, there has not been much work on providing *language support* for variational programming.

The choice calculus (CC) is a formal language for representing *static* variation in trees and other data [23, 50], which is the basis for the choice-based variation representation used in VPC. It provides choices for systematically representing variation, but itself offers no constructs for transforming such representations. There have been a few attempts at extending CC with variational programming features. The compositional choice calculus (CCC) extends CC with functions [51]. It supports generating new variation and a form of variation-preserving computing, but does not provide any mechanisms for eliminating, aggregating, or transforming the variation structure. The main goal of CCC is to unify compositional and annotative approaches to feature implementation, so it also provides a generic mechanism for composing two ASTs. A different extension of CC adds a selection operation to eliminate choices [22]. It focuses mostly on the interaction between selections and a construct for declaring locally-scoped dimensions which we have omitted from VPC, and discusses the impact of the semantics on the modularity of software product line specifications. It does not provide any support for computing with variation.

There have also been several language-based approaches to *compositionally constructing* variational programs with aspects [36], by step-wise refinement [5], or through delta-oriented programming [42]. Each of these provide mechanisms for modularizing changes that may be conditionally applied to a program. As with in-place variation, a substantial amount of work has been done in the software product line community on analyzing and verifying compositionally constructed variational programs. An excellent survey of analyses on both varieties of product line is provided by Thüm et al. [47]. The compositional approach is quite different from our view of in-place variation, so VPC is less applicable in this context. However, one finding of Thüm et al.'s survey is that whole-family analyses are less common over compositional product lines, at least in the literature.

There are a few library-based approaches to variational programming. The concept of variational programming, independent of its applications, was first proposed via a Haskell library [24]. This library requires data types to be explicitly extended by new cases to support variation. Monad instances support convenient variation-preserving computations within a single variational data type, but more advanced use cases—involving multiple variational types or any kind of aggregation or transformation—entail quite some notational overhead to wrap/unwrap variational types and pattern match on variation constructs. This contrasts with VPC where functions are mapped automatically over arbitrary variational structures. TypeChef [29] is a system developed for parsing and type checking `#ifdef`-annotated source code. To do this, it provides a library of data structures and operations to support variational programming. As a library, it suffers similar inconveniences and limitations as [24]; however, it has been successfully employed in a number projects for analyzing software product lines [37, 38, 33, 30, 34]. Variational data structures and idioms to support variational programming are described in [52], most of which are adapted from the above works.

Work on algebraic effects and effect handlers has frequently used a choice as an example of a non-deterministic effect [4, 8, 27, 41]. These choice effects differ from VPC choices in two key ways: (1) Each choice effect is independent—there is no concept analogous to VPC's dimensions to synchronize selections across choices. (2) The evaluation of an expression with choice effects yields an unstructured set of variants, rather than a structured choice that clarifies the relationship between a sequence of selections and the variant it yields, as in VPC. A notable exception is the "selection functional" effect presented in [4], which essentially implements dimensioned choices in the *Eff* programming language. Both choice effects and selection functionals are more restrictive than VPC choices since they do not permit the alternatives of the choice, or the produced variants, to be values of different types. Additionally, the control flow enforced by these encodings of choices rules out several ways to optimize variation-preserving computations. Since computations for different alternatives are performed in different continuations, we lose the opportunity to perform choice reduction to proactively eliminate unreachable alternatives [17], or to join converged execution paths early. Finally, encoding choices as effects loses the ability to explicitly reflect on the structure of a variational expression and perform transformations. This ability is often needed for variational analyses, for example, the ability to commute selections with computations is a critical transformation for analyzing SPLs [29, 28, 46, 2, 17, 16]. All of the limitations of choice effects and selection functionals relative to VPC can likely be overcome through more clever effect encodings. However, this would essentially amount to embedding VPC into a host language with a rather complicated semantics, complicating our understanding of effective variational programming.

Variation-preserving computations can also be encoded as computations in a reader monad, where the environment identifies a single variant (for example, a predicate on dimensions) and choices are encoded as conditional statements querying this environment. Some of the limitations of this encoding echo the above: It does not support alternatives of different types, and it doesn't support optimizations or transformations since we cannot reflect on the structure of variation in the program. The reader-based encoding has the additional drawback of describing the variational results of a computation intensionally, rather than extensionally. That is, the result is a function that can be used to obtain different variants by passing in different environments, rather than an explicit choice structure as obtained from VPC. This makes it prohibitively expensive to enumerate the variants of a result since we must try all possible selections of the dimensions, rather than just iterating over the variants at the leaves of a choice expression.

There are, of course, many applications that do some form of variation programming, but without any specific programming support. The idea behind *multi-execution* is to run small number of program alternatives in parallel; it has been applied to identify security problems [21, 20, 9, 32], configuration bugs [43], and update inconsistencies [49, 26, 35] in programs. Most of these applications synchronize the different executions externally, but some approaches do exploit similarities between programs run in parallel [49, 45]. Most of these approaches are limited to running two programs at a time and would not scale to number of variants encountered in variational programming.

8 Conclusions

Variational programming supports a wide variety of applications that must somehow deal with variation in programs or data, but computing with and transforming variation representations in general-purpose programming languages is effort intensive and error prone. In this paper, we have presented VPC, a core calculus that supports the implementation of a variational functional programming language that provides direct support for variational programming. This language directly supports the implementation of high-level functions for aggregating and transforming variation, as illustrated in Section 2. Additionally, it enables a very simple implementation of a variational type unification algorithm, which is quite complicated without built-in support for variation [17]. As future work we intend to implement other pre-existing variational programming applications using VPC.

We have defined a small-step operational semantics for VPC, and demonstrated that it offers flexibility in its reduction strategy. In existing variational programming applications, much effort is often needed to demonstrate that computations and variation commute properly, but this is built into the semantics of VPC. We presented a type system that relates variational types with VPC expressions. The type system supports dimension polymorphism through a restricted form of dependent types. Finally, we described a type inference algorithm for automatically inferring most general types for VPC expressions. At the core of this algorithm is a new variational constraint solving algorithm.

Acknowledgments

We would like to thank the anonymous reviewers for exceptionally detailed comments that improved the quality of this paper. This work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

References

1 S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein. Domain types: Abstract-domain selection based on variable usage. In *HVC*, LNCS 8244, pages 262–278, 2013.

2 S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *JASE*, 17(3):251–300, 2010.

3 T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, pages 165–178, 2012.

4 Andrej B. and Matija P. Programming with algebraic effects and handlers. *CoRR*, abs/1203.1539, 2012.

5 D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *TSE*, 30(6):355–371, 2004.

6 E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL^{LIFT}: Statically analyzing software product lines in minutes instead of years. In *PLDI*, pages 355–364, 2013.

7 C. Brabrand, M. Ribeiro, T. Tolêdo, and Paulo B. Intraprocedural dataflow analysis for software product lines. In *AOSD*, pages 13–24, 2012.

8 E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*, pages 133–144, 2013.

9 R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *ACSAC*, pages 322–331, 2008.

10 S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL*, pages 583–594, 2014.

11 S. Chen and M. Erwig. Early detection of type errors in C++ templates. In *PEPM*, pages 133–144, 2014.

12 S. Chen and M. Erwig. Guided type debugging. In *FLOPS*, LNCS 8475, pages 35–51. 2014.

13 S. Chen and M. Erwig. Type-based parametric analysis of program families. In *ICFP*, pages 39–51, 2014.

14 S. Chen and M. Erwig. Principal type inference for GADTs. In *POPL*, pages 416–428, 2016.

15 S. Chen, M. Erwig, and K. Smeltzer. Let's hear both sides: On combining type-error reporting tools. In *VL/HCC*, pages 145–152, 2014.

16 S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ICFP*, pages 29–40, 2012.

17 S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. *TOPLAS*, 36(1):1:1–1:54, 2014.

18 S. Chen, M. Erwig, and E. Walkingshaw. A calculus for variational programming. Technical report, University of Louisiana at Lafayette and Oregon State University, 2016. URL: `http://shengchen1.bitbucket.org/ws/TechReport/vpc.pdf`.

19 M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. *TSE*, 34(5):597–613, 2008.

20 W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: A web browser with flexible and precise information flow control. In *CCS*, pages 748–759, 2012.

21 D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *S&P*, pages 109–124, 2010.

22 M. Erwig, K. Ostermann, T. Rendel, and E. Walkingshaw. Adding configuration to the choice calculus. In *VAMOS*, pages 13:1–13:8, 2013.

23 M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *TOSEM*, 21(1):6:1–6:27, 2011.

24 M. Erwig and E. Walkingshaw. Variation programming with the choice calculus. In *GTTSE*, LNCS 7680, pages 55–100, 2013.

25 R. Harper. *Practical Foundations for Programming Languages (2ed)*. Cambridge University Press, New York, NY, USA, 2016.

26 P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *ICSE*, pages 612–621, 2013.

27 O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *ICFP*, pages 145–158, 2013.

28 C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *TOSEM*, 21(3):14:1–14:39, 2012.

29 C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA*, pages 805–824, 2011.

30 C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *FOSD*, pages 1–8, 2012.

31 C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *ISSRE*, pages 221–230, 2012.

32 C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *S&P*, pages 443–457, 2012.

33 J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. Morpheus: Variability-aware refactoring in the wild. In *ICSE*, pages 380–391, 2015.

34 J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *ESEC/FSE*, pages 81–91, 2013.

35 M. Maurer and D. Brumley. TACHYON: Tandem execution for efficient live patch testing. In *USENIX Security Symposium*, pages 43–43, 2012.

36 M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. *SEN*, 29(6):127–136, 2004.

37 S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: Static analyses and empirical results. In *ICSE*, pages 140–151, 2014.

38 S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Where do configuration constraints stem from? An extraction approach and an empirical study. *TSE*, 2015.

39 H. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *ICSE*, pages 907–918, 2014.

40 M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

41 G. Plotkin and J. Power. Adequacy for algebraic effects. In *FoSSaCS*, pages 1–24, 2001.

42 I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*, pages 77–91. 2010.

43 Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *SOSP*, pages 237–250, 2007.

44 M. Sulzmann. A general type inference framework for hindley/milner style systems. In *FLOPS*, pages 248–263, 2001.

45 W. N. Sumner, T. Bao, X. Zhang, and S. Prabhakar. Coalescing executions for fast uncertainty analysis. In *ICSE*, pages 581–590, 2011.

46 S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *GPCE*, pages 95–104, 2007.

47 T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *CSUR*, 47(1):6, 2014.

48 E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541, 2014.

49 J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *ASPLOS*, pages 193–204, 2009.

50 E. Walkingshaw. *The choice calculus: A formal language of variation.* PhD thesis, Oregon State University, 2013.

51 E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. In *GPCE*, pages 132–140, 2012.

52 E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational data structures: Exploring trade-offs in computing with variability. In *Onward!*, pages 213–226, 2014.

53 E. Walkingshaw and K. Ostermann. Projectional editing of variational software. In *GPCE*, pages 29–38, 2014.

A Appendix

A.1 Confluence of Reductions

Proof sketch for Lemma 1. A full proof of confluence proceeds by induction on the structure of e, considering for each term, every possible pair of reductions and constructing a common e''' from the result of each. Most of these cases are not interesting, for example, we can reduce either the **then** or **else** branch of an **any**/**the**-expression, then immediately reduce the other. Throughout this section, we have noted places where we have enforced a particular evaluation order to preserve confluence. For example, the scrutinee must be fully reduced before reducing an **any**/**the**-expression, and we do not allow reduction in the body of abstractions or selections. In this proof sketch, we illustrate a few case involving applications and choices to give a sense for the structure of the full proof.

Consider a function application with a choice on the RHS, $e = (\lambda v.e_1)\ D\langle e_2, e_3\rangle$. We can either (1) apply the App-Reduce rule, or (2) reduce one alternative, say the left, of the choice via the In-App-R and In-Chc-L congruence rules. Path (1) produces a **split** expression as e'. Path (2) produces $e'' = D\langle e_2', e_3\rangle$. From each of these, we can produce $e''' = D\langle(\lambda v.e_1)\ e_2', (\lambda v.e_1)\ e_3\rangle$. Figure 5 shows a reduction sequence along path (1). From here, further reduction depends on the variability of e_2 and e_3. For simplicity, assume both are plain; then two sequences of App-Reduce, *expand* **split**-**any**-**else**, and Any-Reduce-Else will produce the expression $D\langle(\lambda v.e_1)\ e_2, (\lambda v.e_1)\ e_3\rangle$. Now we can produce e''' via the In-Chc-L and In-App-R congruence rules. It is easy to see that the same reduction sequence, minus the final step, produces e''' from e'' since at each step e_2 is simply replaced by e_2'.

Next consider an application where both sides contain a choice in the same dimension, as in $e = D\langle\lambda v.e_1, \lambda v.e_2\rangle\ D\langle e_3, e_4\rangle$. Now we can either (1) apply the App-Chc rule, resulting in $e' = D\langle(\lambda v.e_1)\ (D_\ell \blacktriangleright D\langle e_3, e_4\rangle), (\lambda v.e_2)\ (D_\ell \blacktriangleright D\langle e_3, e_4\rangle)\rangle$, or (2) reduce the left alternative of the choice on the RHS, resulting in $e'' = D\langle\lambda v.e_1, \lambda v.e_2\rangle\ D\langle e_3', e_4\rangle$. From each of these, we can produce $e''' = D\langle(\lambda v.e_1)\ e_3', (\lambda v.e_2)\ e_4\rangle$. From e', we apply Chc-Elim-L via In-Chc-L, then Chc-Elim-R via In-Chc-R, and finally reduce e_3 via In-Chc-L, In-App-R. From e'', we apply App-Chc followed by Chc-Elim-L via In-Chc-L and Chc-Elim-R via In-Chc-R. □

A.2 Congruence rules

Figure 12 presents the full set of congruence rules for the operational semantics of vpc. These rules are discussed in Section 3.2.

A.3 Constraint generation rules

Figure 13 presents the full set of constraint generation rules to support type inference in vpc. Constraint generation is discussed in Section 5.

A.4 Relationship Between the Constraint Solver and VLC

In VLC [17], a variational unification algorithm was developed for unifying variational types. Although the types contain dimension literals only, the algorithm was quite complicated in its way of distinguishing type variables that appear in different alternatives but share the same name. To see why we need to distinguish type variables with the same name, let's consider the type equivalence constraint $A\langle\texttt{Int}, \alpha\rangle \equiv A\langle\alpha, \texttt{Bool}\rangle$. For this constraint, our first attempt was to decompose it into two subproblems $\texttt{Int} \equiv \alpha$ and $\alpha \equiv \texttt{Bool}$, which are trivial to

$$\textsc{In-App-L}\quad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \textsc{In-App-R}\quad \frac{e_2 \longrightarrow e_2'}{e_1\ e_2 \longrightarrow e_1\ e_2'} \qquad \textsc{In-Chc-L}\quad \frac{e_1 \longrightarrow e_1'}{D\langle e_1, e_2\rangle \longrightarrow D\langle e_1', e_2\rangle} \qquad \textsc{In-Chc-R}\quad \frac{e_2 \longrightarrow e_2'}{D\langle e_1, e_2\rangle \longrightarrow D\langle e_1, e_2'\rangle}$$

$$\textsc{In-Let-Bound}\quad \frac{e_1 \longrightarrow e_1'}{\textbf{let}\ v = e_1\ \textbf{in}\ e_2 \longrightarrow \textbf{let}\ v = e_1'\ \textbf{in}\ e_2} \qquad \textsc{In-Let-Body}\quad \frac{e_2 \longrightarrow e_2'}{\textbf{let}\ v = e_1\ \textbf{in}\ e_2 \longrightarrow \textbf{let}\ v = e_1\ \textbf{in}\ e_2'}$$

$$\textsc{In-Any-From}\quad \frac{e_1 \longrightarrow e_1'}{\textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 \longrightarrow \textbf{any}\ d\ \textbf{from}\ e_1'\ \textbf{in}\ e_2\ \textbf{else}\ e_3}$$

$$\textsc{In-The-From}\quad \frac{e_1 \longrightarrow e_1'}{\textbf{the}\ \delta\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 \longrightarrow \textbf{the}\ \delta\ \textbf{from}\ e_1'\ \textbf{in}\ e_2\ \textbf{else}\ e_3}$$

$$\textsc{In-Any-Then}\quad \frac{e_2 \longrightarrow e_2'}{\textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 \longrightarrow \textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2'\ \textbf{else}\ e_3}$$

$$\textsc{In-The-Then}\quad \frac{e_2 \longrightarrow e_2'}{\textbf{the}\ \delta\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 \longrightarrow \textbf{the}\ \delta\ \textbf{from}\ e_1\ \textbf{in}\ e_2'\ \textbf{else}\ e_3}$$

$$\textsc{In-Any-Else}\quad \frac{e_3 \longrightarrow e_3'}{\textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 \longrightarrow \textbf{any}\ d\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3'}$$

$$\textsc{In-The-Else}\quad \frac{e_3 \longrightarrow e_3'}{\textbf{the}\ \delta\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3 \longrightarrow \textbf{the}\ \delta\ \textbf{from}\ e_1\ \textbf{in}\ e_2\ \textbf{else}\ e_3'}$$

Figure 12 Small-step operational semantics of VPC: congruence rules.

solve with the solutions $\alpha \mapsto \texttt{Int}$ and $\alpha \mapsto \texttt{Bool}$, respectively. Putting the solutions together requires mapping α to two different types, yielding a conflict.

VLC addresses this problem by first qualifying type variables with the locations they appear. Qualification brings the original constraint to $A\langle \texttt{Int}, \alpha_{A.2}\rangle \equiv A\langle \alpha_{A.1}, \texttt{Bool}\rangle$ VLC then decomposes this new constraint, solves subproblems, and puts the solutions together, yielding the solution $\{\alpha_{A.1} \mapsto \texttt{Int}, \alpha_{A.2} \mapsto \texttt{Bool}\}$. To transform a solution for the qualified problem to a solution for the original problem, VLC uses a step called completion. Completion turns each mapping $\alpha_q \mapsto \phi$ into a mapping $\alpha \mapsto \phi'$ so that ϕ' has ϕ in the location q. Moreover, there should be only one ϕ' for each type variable after completion. Applying completion to our example yields the solution $\alpha \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle$.

While types in this paper contain dimension literals as well dimension variables, constraint solving in Figure 11 doesn't need variable qualification for two reasons. First, as explained in Section 6.1 and implemented in Figure 11, unification of types containing dimension variables are deferred until all dimension variables are instantiated with dimension literals. Second, as can be seen from case (d) in Figure 11, our constraint solver uses *vunify*, an improved version of the unification algorithm used in VLC, which doesn't need qualifications and completion.

The key insight that has allowed the simplification in *vunify* is that the need for qualification arises because a type variable used in different alternatives received its binding from a global context that can only provide one value for it. For example, when solving the two subproblems $\texttt{Int} \equiv \alpha$ and $\alpha \equiv \texttt{Bool}$ from $A\langle \texttt{Int}, \alpha\rangle \equiv A\langle \alpha, \texttt{Bool}\rangle$, a name conflict occurs for α because there is no way to assign a single type to α in a global context. This conflict can be avoided if we divide the global context into sub-contexts and perform constraint solving in the individual sub-contexts. In our example, we can split the global context into two contexts named $A.1$ and $A.2$ and solve constraint $\texttt{Int} \equiv \alpha$ in $A.1$ and $\alpha \equiv \texttt{Bool}$ in $A.2$. This allows us to map α to different types in the different contexts. The solutions from split contexts can be merged into a solution for the global context if we respect the ordering of the contexts. For example, the result from the context $A.1$ will appear in the first alternative of A only. Following this idea, the solution $\alpha \mapsto \texttt{Int}$ from $A.1$ and $\alpha \mapsto \texttt{Bool}$ from $A.2$ can be combined to $\alpha \mapsto A\langle \texttt{Int}, \texttt{Bool}\rangle$, which is indeed the most general unifier to the unification problem $A\langle \texttt{Int}, \alpha\rangle \equiv A\langle \alpha, \texttt{Bool}\rangle$.

$$\textsc{I-Con}\quad \dfrac{\kappa \text{ of type } \gamma \qquad \alpha \text{ fresh}}{\gamma \equiv \alpha; \Gamma; \Delta \vdash_I \kappa : \alpha} \qquad\qquad \textsc{I-Var}\quad \dfrac{\Gamma(v) = \forall \overline{\alpha}\overline{d}.C \Rightarrow \phi \qquad \alpha \text{ fresh}}{\exists \overline{\alpha}\overline{d}.(C \wedge \phi \equiv \alpha); \Gamma; \Delta \vdash_I v : \alpha}$$

$$\textsc{I-Abs}\quad \dfrac{C; \Gamma, (v, \alpha_1); \Delta \vdash_I e : \alpha_2 \qquad \alpha_3 \text{ fresh}}{\exists \alpha_1 \alpha_2.(C \wedge \alpha_3 \equiv \alpha_1 \to \alpha_2); \Gamma; \Delta \vdash_I \lambda v.e : \alpha_3}$$

$$\textsc{I-App}\quad \dfrac{C_1; \Gamma; \Delta \vdash_I e_1 : \alpha_1 \qquad C_2; \Gamma; \Delta \vdash_I e_2 : \alpha_2 \qquad \alpha_3 \text{ fresh}}{\exists \alpha_1 \alpha_2.(C_1 \wedge C_2 \wedge \alpha_1 \equiv \alpha_2 \to \alpha_3); \Gamma; \Delta \vdash_I e_1\ e_2 : \alpha_3}$$

$$\textsc{I-AbsD}\quad \dfrac{C; \Gamma; \Delta, (d, d) \vdash_I e : \alpha_2 \qquad \alpha_3 \text{ fresh}}{\exists \alpha_2 d.(C \wedge \alpha_3 \equiv d \to \alpha_2); \Gamma; \Delta \vdash_I \lambda d.e : \alpha_3}$$

$$\textsc{I-Let}\quad \dfrac{\begin{matrix} \alpha \text{ fresh} & C_1; \Gamma, (v, \alpha); \Delta \vdash_I e_1 : \alpha_1 \\ C_2 = C_1 \wedge \alpha \equiv \alpha_1 & C_3; \Gamma, (v, \forall \alpha_1.C_2 \Rightarrow \alpha_1); \Delta \vdash_I e_2 : \alpha_2 \end{matrix}}{(\exists \alpha \alpha_1.C_2) \wedge C_3; \Gamma; \Delta \vdash_I \textbf{let } v = e_1 \textbf{ in } e_2 : \alpha_2}$$

$$\textsc{I-Sel}\quad \dfrac{\delta \in \Delta \qquad C_2 = C_1 \wedge \alpha \equiv \delta\langle \alpha_\ell, \alpha_r \rangle \qquad \begin{matrix} C_1; \Gamma; \Delta \vdash_I e_1 : \alpha \\ \begin{cases} C_3 = C_2 \wedge \alpha_1 \equiv \alpha_\ell & s = \ell \\ C_3 = C_2 \wedge \alpha_1 \equiv \alpha_r & s = r \end{cases} \end{matrix} \qquad \alpha_\ell, \alpha_r, \alpha_1 \text{ fresh}}{\exists \alpha \alpha_\ell \alpha_r.C_3; \Gamma; \Delta \vdash \delta_s \blacktriangleright e : \alpha_1}$$

$$\textsc{I-Dim}\quad \dfrac{\alpha \text{ fresh}}{\delta \equiv \alpha; \Gamma; \Delta \vdash_I \delta : \alpha} \qquad \textsc{I-Chc}\quad \dfrac{C_1; \Gamma; \Delta \vdash_I e_1 : \alpha_1 \qquad C_2; \Gamma; \Delta \vdash_I e_2 : \alpha_2 \qquad \alpha_3 \text{ fresh}}{\exists \alpha_1 \alpha_2.(\delta\langle C_1, C_2 \rangle \wedge \delta\langle \alpha_1, \alpha_2 \rangle \equiv \alpha_3); \Gamma; \Delta \vdash_I \delta\langle e_1, e_2 \rangle : \alpha_3}$$

$$\textsc{I-Any}\quad \dfrac{C_1; \Gamma; \Delta \vdash_I e_1 : \alpha_1 \qquad C_2; \Gamma; \Delta, (d, d) \vdash_I e_2 : \alpha_2 \qquad C_3; \Gamma; \Delta \vdash e_3 : \alpha_3}{\exists \alpha_1 \alpha_2 d.(C_1 \wedge C_2 \wedge C_3 \wedge \alpha_2 \equiv \alpha_3 \wedge (d \in \alpha_3 \Rightarrow d{\downarrow}\alpha_1)); \Gamma; \Delta \vdash_I \textbf{any } d \textbf{ from } e_1 \textbf{ in } e_2 \textbf{ else } e_3 : \alpha_3}$$

$$\textsc{I-The}\quad \dfrac{C_1; \Gamma; \Delta \vdash_I e_1 : \alpha_1 \qquad C_2; \Gamma; \Delta \vdash_I e_2 : \alpha_2 \qquad C_3; \Gamma; \Delta \vdash e_3 : \alpha_3 \qquad \Delta(\delta) = \delta_1}{\exists \alpha_1 \alpha_2.(C_1 \wedge C_2 \wedge); \Gamma; \Delta \vdash_I \textbf{the } \delta \textbf{ from } e_1 \textbf{ in } e_2 \textbf{ else } e_3 : \phi_3}$$

Figure 13 Full set of constraint generation rules for VPC