# Variational Data Structures: Exploring Tradeoffs in Computing with Variability


Eric Walkingshaw
University of Marburg
Germany

Christian Kästner
Carnegie Mellon University
USA

Martin Erwig
Oregon State University
USA

Sven Apel
University of Passau
Germany

Eric Bodden
Fraunhofer SIT & TU Darmstadt
Germany



## Abstract

Variation is everywhere, and in the construction and analysis of customizable software it is paramount. In this context, there arises a need for *variational data structures* for efficiently representing and computing with related variants of an underlying data type. So far, variational data structures have been explored and developed ad hoc. This paper is a first attempt and a call to action for systematic and foundational research in this area. Research on variational data structures will benefit not only customizable software, but many other application domains that must cope with variability. In this paper, we show how support for variation can be understood as a general and orthogonal property of data types, data structures, and algorithms. We begin a systematic exploration of basic variational data structures, exploring the tradeoffs among different implementations. Finally, we retrospectively analyze the design decisions in our own previous work where we have independently encountered problems requiring variational data structures.



***Categories and Subject Descriptors*** E.1 [*Data Structures*]

***General Terms*** Design, Performance, Theory

***Keywords*** variation, data structures, configurable software, software product lines, variability-aware analyses


## 1. Introduction

> Variability is the law of life, and as no two faces are the same, so no two bodies are alike, and no two individuals react alike and behave alike . . . [39]

Although referring to a special class of "biological systems", this quote of William Osler, one of the icons of modern medicine, captures the pervasiveness of variability. In almost all domains, variation is the rule, from biological and social to economic and technical systems. Software is no exception.

A simple example of variation in software is parameterizing a program to support different use cases. To execute the program and compute a result, we must select a particular configuration of parameters. To compute results for other configurations, we must repeatedly configure and run the program. This serial approach to computing alternatives obscures variability in the problem we are trying to solve and performs redundant work that may have been shared among alternative configurations. The solution is to lift this variation into the program itself; that is, to compute with *variational data* explicitly, rather than with each alternative separately.

To illustrate, suppose we are planning a trip from Frankfurt Airport to Schloss Dagstuhl. Some travel planning software finds that there are many possible itineraries: taking a train then a bus; taking a train to one of two different stations, then a taxi; or taking a taxi the whole way. Now suppose we want to compute the price of the trip per person, which for the taxi also depends on the number of people traveling together. Traditionally, we might write a function of type `(Itinerary,GroupSize) => Cost`.[1] This function is variational in the sense that it supports different inputs, but whenever we use it, we must select a particular itinerary and group size. Computing the costs of all travel options requires calling the function multiple times, once for each configuration. This is wasteful since we need to repeat (or cache) many of the same calculations; for example, we need to repeatedly lookup the price of the same train connections even if they do not change based on other parts of the itinerary or the group size.

Instead, we envision a function of type `(V[Itinerary], V[GroupSize]) => V[Cost]`, that accepts a *variational* itinerary and a *variational* group size as input, and *efficiently* computes the *variational* results. In other words, we pass in all alterna-

[1] We use Scala syntax for types and code examples.



*Onward! Papers '14*, October 20–24, 2014, Portland, Oregon, USA.



http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

tives of interest and get back the costs for all of them at once. The variational inputs and outputs are not just sets of alternatives, but are *structured* such that we can see exactly which cost goes with which combination of itinerary and group size. By working with variational data rather than computing the results for each alternative sequentially, the algorithm can reuse work and exploit shared components more easily.

*Variational data structures* are data structures for efficiently representing and computing with variational data. For example, a variational itinerary data structure should share the common sub-paths of alternative itineraries. In this way, variational data structures support the definition of efficient variation-preserving or variability-aware algorithms [43].

The overarching goal of this paper is to promote foundational research on variational data structures. We want to raise awareness that variability can be dealt with systematically, and that many existing problems can benefit by considering the tradeoffs among well-understood variational data structures rather than implementing ad hoc solutions. This paper is a call to action for a systematic exploration of this design space by a group of authors who have separately run against the need to manage variational data, and who have, so far, themselves explored it in only a need-driven way.

In pursuit of this goal: (1) We argue in Section 2 that research on variational data structures is important and has a broad range of applications, both in software engineering and beyond, many of which may not be initially obvious. (2) We provide a baseline for discussion and future research by enumerating some basic variation representations in Section 3, providing a set of initial variational data structures in Section 4, and describing basic techniques for computing with variability in Section 5. (3) We demonstrate in Sections 4 and 6 that the design decisions and tradeoffs regarding variational data structures are non-obvious. As with traditional data structures, different implementations of variational data structures better support different use cases, with space and runtime efficiency tradeoffs among them.

Over the course of the paper, we make the following concrete contributions:

1. An abstract model of variational values as choices between labeled alternatives (Section 3.1), and three instantiations of this model (Section 3.2) based on our own previous work on the choice calculus [21] and TypeChef [31].
2. Some examples of variational data structures (Section 4), including variational lists, variational maps, and variational sets. For lists, we explore several alternative implementations and discuss the tradeoffs among them.
3. A retrospective case-study analysis of design decisions regarding variational data in our own previous work (Section 6). Using the types and data structures defined in the first two contributions, we discuss the tradeoffs among alternative implementations, and how a more systematic view of variational data structures would have helped us make more informed decisions.

We use our own previous work for the case study analysis since we have immediate access to the history and design rationale of these projects. In Section 7, we discuss other encounters with variational data structures and related efforts in the community to cope with variation.

Note that the example variational data structures presented in this paper represent a tiny fraction of the design space in this area. There are both many more possible implementations of the variational data structures we discuss, and many other data structures that can be adapted to support variability.

## 2. Motivation

Variation is an important dimension of complexity. To achieve efficient computation in the presence of variation, we must represent input, intermediate results, and output in a compact variational form. Since variational output from one function can be used as variational input to the next, this view leads to ubiquitous variability in data structures and algorithms.

In this section, we provide a number of motivating examples for variational data structures. Since these examples are quite diverse, we also set the scope of the paper by characterizing the kind of variation we consider in terms of functions from variational inputs to variational outputs.

### 2.1 Motivating Examples

***Variational Program Analyses.*** Most static program analyses operate on data structures extracted from program syntax, such as maps (e.g. symbol tables), lists (e.g. wait lists), trees (e.g. abstract syntax trees), and graphs (e.g. data-flow and control-flow graphs). Recently, researchers have begun to *lift* program analyses to *variational programs*, such as software product lines [43], which can be configured to generate many different variant programs. This lifting affects both the internal data structures and the analysis algorithms.

For example, the lifted abstract syntax tree (AST) of a variational program represents the AST of all possible variants. Variational ASTs have been used to implement variational type checking [2, 30, 32, 36] and variational type inference [13, 14], which ensure the static type safety of a variational program as a whole, simultaneously covering all of the variants that can be generated. Variational type systems take as input the variational AST and produce as output the variational typing result that indicates which variants are well-typed. Likewise, variational control-flow graphs have been used to lift data-flow and model-checking analyses to variational programs [5, 9, 11, 16, 35, 36]. By exploiting sharing among variants encoded in the variational AST or variational graph, lifted analyses are able to efficiently verify properties for all variants at once, whereas verifying each variant individually is usually intractable.

Besides the variational representation of the program itself, variational program analyses must store intermediate data in all kinds of variational data structures, including variational symbol tables, program-state maps, and parameter lists. The explicit representation of variation within a shared context is the key to supporting efficient variational analyses [5, 11, 36].

***Variational Software Artifacts.*** Besides source code, other kinds of software artifacts may exhibit variation, such as documentation and test suites, and many variation representations are sufficiently general to support this [1, 3, 8]. Thüm et al. [44] proposed incorporating variation in test suites and formal specifications, which supports the development of new kinds of variational analyses, for example, variational test execution [6, 33, 34, 37, 45] and variational deductive verification [44]. Variational test execution, which entails simulating the execution of a test on all software variants, needs substantial support from variational data structures since arbitrary computation must be performed in a variational setting.

There are many opportunities for future work in representing and analyzing variation in supplementary software artifacts, such as variational document analysis, performance modeling, build-system analysis, and so on, all of which require support from variational data structures.

***Beyond Variational Software.*** In principle, any parameterized operation can be made variational to simultaneously consider several alternative inputs/configurations at once. The travel planning example in Section 1 is just one illustration of the potential of this view.

There are also many applications that *already* cope with variation. Since there is no underlying theory, they usually do this in an ad hoc way. In our travel planning example, we took as a given a route-planning service that returned a list of alternative itineraries. This service must cope with variation in several different kinds of data: The transportation network graph may be variational [23] if users can choose what modes of transportation they are willing to take, how far they are willing to walk, etc. The input may be variational if we allow for optional waypoints (nodes in the graph) or prioritize certain modes of travel (edges). Finally, the output is a variational path through the graph, and may also return variational properties of the paths, such as time and cost.

Variational data structures can also be used to simulate computation in the presence of uncertainty. For example, Chen et al. have shown how variational types can be used to recover from type errors during type inference [13] and improve the quality of type errors [12]. Similarly, variational data structures can support the efficient simulation of all possibilities in alternative programming models, such as probabilistic computing; to track context information that controls an algorithm, as in context-oriented programming [26]; or to maintain alternative views of sensitive values corresponding to different privacy policies [6, 7, 46].

### 2.2 Scope

The motivating examples illustrate the diversity of applications for variational data structures. Since "variation" is a very broad term, in this section we set the scope of this paper by clarifying the kind of variation that we consider.

Consider a function with multiple arguments. A function already offers variability in that it can be evaluated with different arguments, but this is *not* the variability we address in this paper. Likewise, there may be variation in the algorithm computed by a function; for example, a function may sort a list using either merge sort or ring sort depending on the setting of a configuration option. Such configuration options induce a certain variability, but as long as we pick a specific configuration before executing the function, the execution itself is not variational, and so is not the focus of this paper.

Instead, we focus on *variation-preserving functions*, which take one or more variational inputs and produce *correspondingly* variational output. A variational input or output can have a *finite* number of values *at the same time* during computation. In other words, a variation-preserving function computes the result for all combinations of alternative argument values, generally leading to several alternative results. Additionally, variational inputs and outputs are structured such that the relationship of each output to its corresponding inputs is clear. For example, in the travel-planning scenario, our function accepts two variational inputs, for comparing different itineraries and group sizes, and computes a variational output representing the cost corresponding to each combination of inputs.

Functions with static configuration options (e.g. implemented by C preprocessor directives) can be transformed into regular functions by promoting the configuration options to function arguments. This process, which has been described similarly elsewhere [40], is illustrated in pseudocode below.

```
createFile(name) {          createFile(os, name) {
#if OS == "Windows"           if (os == "Windows") {
  ...                           ...
#elif OS == "Unix"     ⇝      } else if (os == "Unix") {
  ...                           ...
#endif                        }
}                           }
```

Now the `createFile` function can be made variation-preserving by interpreting the first argument (`os`) as a variational string. Similar translations are possible for other compile-time variability mechanisms [4], such as feature-module composition and aspect weaving. For example, a choice between two function implementations can be translated into a new function with an additional variational argument. The issue of different binding times (run-time vs. load-time vs. compile-time variability) is orthogonal to this model. The key is that there are some arguments that are not bound to a single value when calling the function, but to *multiple values*.

## 3. Variational Data Types

In this section, we develop an abstract model of variation in data types and three ways to implement it. The model is flat: it describes variability in terms of a mapping from configurations to plain (non-variational) variants. This is obviously inefficient for representing variation in composite data types since common subparts will not be shared. In Section 4, we show how to incorporate more sophisticated mechanisms to support various kinds of sharing. The notations and concepts developed in this section will provide a framework for describing and comparing variational data structures.

### 3.1 Choices as Variational Values

A variational value `v` exists in the context of a *configuration space* `C`, which describes all possible ways that `v` can be configured. Abstractly, we can represent `C` as a finite set of *configurations*. For example, if `C` is defined by a feature model [28], then each configuration may be represented by a set of features to include, and the set `C` contains all sets of features that are consistent with the feature model. However, our model is independent of any particular encoding of `C`.

The meaning of `v` is defined in terms of its resolution to a *plain* value for every configuration in `C`. A plain value is one that does not contain variation. We call the set of plain values that `v` can be resolved to its *variants*, which must all be of the same type `A`. Therefore, the semantics of a variational value `v` is a mapping from its configurations to its variants, `C => A`.

Since the set of configurations is finite, an initial syntactic representation of `v` could be to just represent the mapping from configurations to variants directly. A *choice* is a set of configuration-labeled variants. For example, if `C = {c1,c2,c3}` and `A = Int`, then the choice `<c1:4, c2:4, c3:5>` defines a variational integer where configurations `c1` and `c2` are mapped to the variant `4`, and `c3` is mapped to `5`. However, this representation of choices is inefficient since usually many configurations will map to the same variants. The redundancy can be reduced by recognizing that variation is not arbitrary but corresponds to structure in the configuration space. For example, if `C` is defined by a feature model, then the differences at one variation point, represented by a choice, are likely to correspond to only a small subset of the features.

Therefore, rather than labeling variants by configurations directly, it is more efficient to label them by higher-level concepts in the structure of the configuration space. For example, if configurations `c1` and `c2` contain features `f1` and `f2`, respectively, and `c3` contains neither feature, then we can represent our variational integer as `<f1∨f2: 4, ¬f1∧¬f2: 5>`. In this case, each label describes the subset of the configuration space, or *partial configuration space*, that satisfies the corresponding inclusion condition.

To support our semantic goal of uniquely mapping every configuration to a variant, the set of labels used in any choice `v` must define a *partitioning* of `C`; that is, the partial configuration space defined by each label must be a subset of `C` such that all subsets are pairwise disjoint and the union of all subsets is `C`. We call this invariant on choices the *choice-as-partition* invariant.

### 3.2 Implementing Variational Data Types

A variational value is the simplest kind of variational data structure and can be implemented in a number of different ways. In this section we present three different implementations of variational values that are *generic* in the type of the variants. In Section 4, we will discuss how to weave variability into the definition of *specific* data structures. This will generally lead to different implementations of one variational data type that are tailored to support different usage scenarios.

We present the implementations using Scala code. In fact, the shown data structures have been employed and tested in previous projects (reported in Section 6). Note that, although we use Scala as the metalanguage, the discussion is not tied to Scala in any significant way.

#### 3.2.1 Tag Trees

Whenever the configuration space can be characterized by a set of binary configuration tags, say of type `Tag`, we can employ a binary-tree representation with tags at internal nodes and values stored in leaves. In Scala, we implement this data structure using case classes since these support the use of pattern matching in function definitions. The type parameter `A` represents the (arbitrary) type of values that are made variational by applying the type constructor `V`.

```
trait V[A]
case class One[A](v:A)                        extends V[A]
case class Chc[A](t:Tag, y:V[A], n:V[A]) extends V[A]
```

The class `One` encodes a variant at a leaf in the tree, while `Chc` divides the configuration space into two partial configuration spaces, one in which the tag `t` is selected (`y` for *yes*), and one in which it is deselected (`n` for *no*).

We also introduce syntactic sugar for concisely expressing tag trees. The notation is introduced by example below.[2]

```
T<a,b>        ≡ Chc(T, One(a), One(b))
T<a,U<b,c>> ≡ Chc(T, One(a), Chc(U, One(b), One(c)))
```

By nesting choices within other choices, we can hierarchically divide the configuration space based on the selected and deselected tags. A key benefit of this implementation is that it maintains the choice-as-partition invariant (see Section 3.1) by construction.

When working with variational values, we often need to apply a function to all of the variants it contains. To support this, we introduce two "variational map" operations, which correspond to Scala's idiomatic `map` and `flatMap` methods.[3]

The `map` operation applies a function of type `A => B` to every variant of a variational value of type `V[A]`, preserving its structure. Its implementation for tag trees is shown below.

```
def map[A,B](v: V[A], f: A => B): V[B] = v match {
  case One(v)     => One(f(v))
  case Chc(t,y,n) => Chc(t, map(y,f), map(n,f))
}
```

For example, suppose we have a variational value `x = T<3,4>`, and we want to test whether the variants of `x` are even by applying a function `even` of type `Int => Boolean`. We can apply `even` to all of the variants with `map`, writing `map(x,even)`, which produces the result `T<false,true>`.

---

[2] The notation is based on the choice calculus [21], see Section 6.5.

[3] In Haskell, these are `fmap` and monadic bind `(>>=)`, respectively.

The `flatMap` operation applies a function of type `A => V[B]` (that is, it maps plain `A` values to *variational* `B` values) to every variant of a variational value of type `V[A]`. However, since we want our result to still be of type `V[B]`, we must also "flatten" the resulting variation structure. Its implementation for tag trees is shown below.

```
def flatMap[A,B](v: V[A], f: A=>V[B]): V[B] = v match {
  case One(v)     => f(v)
  case Chc(t,y,n) => Chc(t, flatMap(y,f), flatMap(n,f))
}
```

Consider a function `f` of type `Int => V[Int]` that maps an integer `z` to a choice `U<z,z+5>`. Mapping `f` over `x = T<3,4>` should yield `T<U<3,8>,U<4,9>>`. However, if we apply `map(x,f)`, we get an expression of the wrong type, `V[V[Int]]`.

```
Chc(T, One(Chc(U, One(3), One(8))),
       One(Chc(U, One(4), One(9))))
```

The problem is that trees representing the inner choices are embedded in the leaves of the outer tree. By instead applying `flatMap(x,f)`, we obtain a value of the desired type, `V[Int]`.

```
Chc(T, Chc(U, One(3), One(8))),
       Chc(U, One(4), One(9))))
```

Now the nested choice is represented as a single tag tree with two levels of `Chc` nodes.

The tag-tree representation is based on a hierarchical decomposition of the configuration space using atomic tags. This involves some tradeoffs. The major benefits of this representation are its simplicity and that it preserves the choice-as-partition invariant by construction. The biggest drawback is that certain common partitions of the configuration space cannot be expressed without redundancy. To illustrate with our travel itinerary example: Suppose the 8pm and 10pm trains from Frankfurt Airport both cost 39 EUR whereas missing those trains costs 259 EUR for a taxi. Unfortunately, expressing the variational cost of this part of the itinerary requires repeating the cost of a train ticket, `8pm<39, 10pm<39, 259>>`. The problem can be avoided if we replace the tree representation by a DAG (for example, implemented by a multi-terminal BDD), but this complicates many operations. Another potential drawback of tag trees is that the size of the representation can depend on the order that tags occur in the tree.

#### 3.2.2 Formula Trees

A more flexible representation than tag trees is to use Boolean formulas in choices rather than tags. This enables us to efficiently represent the example at the end of Section 3.2.1 by creating a choice with the formula `8pm ∨ 10pm`. A formula represents a partial configuration space. We can reason about partial configuration spaces by applying SAT solvers to the corresponding formulas.

To implement formula trees, we just replace the `Chc` case of class `V` with a case for choices over formulas.

```
case class Chc[A](m: Formula, y: V[A], n: V[A])
  extends V[A]
```

The implementation of `map` and `flatMap` are identical for tag and formula choices.

With formulas in internal nodes we can reduce redundancy in the representation and still enforce the choice-as-partition invariant by construction. The drawback is that even simple operations that are trivial in tag trees, such as identifying "dead" alternatives that do not correspond to any configurations, require solving satisfiability problems in formula trees.

#### 3.2.3 Formula Maps

A different implementation of `V` employs an internal map data structure from variants to formulas. The formula describes the partial configuration space in which the variant occurs.

```
case class V[A](data: Map[A,Formula]) { ... }
```

The `map` operation, of type `(V[A], A => B) => V[B]`, can be implemented for formula maps by mapping over the keys of the internal `Map` data structure. However, a complication is that if two `A` values map to the same `B` value, we must combine their formulas with a disjunction in the resulting map. For example, suppose we map the function `even` over the formula map `x = V(Map(2 -> i, 3 -> j, 4 -> k))`. Since both `2` and `4` are even, then `map(x, even)` combines these variants to produce the formula map `V(Map(true -> i ∨ k, false -> j))`.

For the `flatMap` operation, of type `(V[A], A => V[B]) => V[B]`, an entry `a -> m` in the initial internal map can expand into potentially many entries of the form `b -> m ∧ n` where each `b` and `n` correspond to entries in the formula map returned by `f(a)`. For example, `flatMap(x, (a:Int) => V(Map(a -> l, a+5 -> ¬l)))` produces the following formula map.

```
V(Map(2 -> i ∧ l, 7 -> i ∧ ¬l,
      3 -> j ∧ l, 8 -> j ∧ ¬l,
      4 -> k ∧ l, 9 -> k ∧ ¬l))
```

Once again, we must potentially merge entries in the result with disjunctions, further complicating the process.

The key advantage of a formula map is that, since values are used as keys, there is exactly one entry for each variant and therefore no redundancy. In a sense, this is a space-optimal representation. In contrast, the tree-based representations do not automatically join redundant variants, which means additional join algorithms have to be implemented [5]. While joins between two siblings are straightforward, joins across different levels of the tree are only possible in formula trees and require nontrivial implementations.

The main disadvantage is that the choice-as-partition invariant is more costly to maintain. For example, every time we add a variant to a formula map, we must update the formulas for every other variant to ensure that their partial configuration space does not overlap with the new variant. It also shares the disadvantage with formula trees of needing to solve satisfiability problems for many operations.

```
def prune[A](v:V[A], sel:Set[Tag], des:Set[Tag]): V[A]
  = v match {
    case One(a) => One(a)
    case Chc(t, yes, no) =>
      if (t ∈ sel) yes
      else if (t ∈ des) no
      else Chc(t, prune(yes, sel ∪ {t}, des),
                  prune(no,  sel, des ∪ {t}))
}
```

**Figure 1.** Pruning dead alternatives from tag trees.

```
def prune[A](v: V[A], ctx: Formula): V[A]
  = v match {
    case One(a) => One(a)
    case Chc(f, yes, no) =>
      if (!SAT(ctx ∧ f)) no
      else if (!SAT(ctx ∧ ¬f)) yes
      else Chc(f, prune(yes, ctx ∧ f),
                  prune(no,  ctx ∧ ¬f))
}
```

**Figure 2.** Pruning dead alternatives from formula trees.

### 3.3 Representation Tradeoffs

The three implementations of variational values presented in this section each have their own strengths and weaknesses. To briefly summarize the discussion so far: Tag trees and formula trees preserve the choice-as-partition invariant by construction; this is costly to maintain in formula maps. On the other hand, formula maps trivially ensure space optimality; this requires costly join operations in formula trees and is impossible in tag trees. Finally, operations on formula maps and formula trees require reasoning about satisfiability problems, typically with external tools, such as SAT solvers or BDDs; operations on tag trees rely only on reasoning about the sets of selected and deselected tags.

This last tradeoff is illustrated by considering, for each representation, the implementation of a function `prune` that removes dead alternatives that are not used in any configuration. For example, the value `2` is dead in `T<T<1,2>,3>` since the selection of tag `T` in the outer choice implies the selection of tag `T` in the inner choice, so `2` can never be chosen. In Figure 1, we implement `prune` for tag trees as a simple recursive function that tracks selected and deselected tags. Dead branches are detected by checking whether a tag is already contained in one of the sets. In contrast, the implementation of `prune` for formula trees in Figure 2, must keep track of a formula representing the partial configuration space for each subtree. Dead branches are detected by performing satisfiability checks—unsatisfiable formulas do not represent any configurations. In formula maps, the operation does not recurse but simply removes all entries with unsatisfiable formulas.

```
def prune[A](v: V[A]): V[A]
  = new V(v.data.filter { case (a,m) => !SAT(m) })
```

Although modern SAT solvers are very efficient even for big formulas, the underlying problem is NP-complete and does not scale well for certain formula shapes.

## 4. Variational Data Structures

In the previous section, we introduced three implementations of a generic `V` type constructor that can produce from any data type `A` a corresponding variational data type `V[A]`. Although convenient, these implementations are inefficient for representing variability in complex data structures since they do not take the internal structure of `A` into account and therefore cannot support the sharing of common subparts. As a simple example, suppose we want to represent the variation between two lists, `[1,2,3,4]` and `[1,2,5,4]`. Naively, we can represent this as a choice between lists, `T<[1,2,3,4],[1,2,5,4]>`, which has type `V[List[Int]]`. This representation repeats shared parts of the two data structures. Simple operations, such as retrieving the first element of the list, require inspecting both alternatives even though they share the same initial sequence of values. Considering only variation among atomic data types introduces a lot of redundancy of this kind, especially when large data structures differ only in minor details.

Therefore, instead of managing variation externally, we must identify ways of pushing variation into the data structure itself to support internal sharing among variants. There are many possible ways to do this. In this section, we provide some examples of variational data structures. Specifically, we present three different implementations of variational lists, a variational map, and a variational set.

These variational data structures are just the tip of the iceberg. Although independently useful, these examples are intended to demonstrate the existence of design tradeoffs among variational data structures. They also provide reference points for the retrospective analysis of our own previous work in Section 6. There is a huge amount of unexplored space both in alternative implementations of the variational data structures we discuss and in variational implementations of the many data structures we do not consider here.

### 4.1 Variational Lists

Let us begin with a traditional non-variational interface for immutable linked lists, parameterized by the type `A` of its elements:[4]

```
trait List[A]
case class Nil[A]()                   extends List[A]
case class Cons[A](h: A, t: List[A]) extends List[A]
def get[A](l: List[A], i: Int): Option[A]
def length[A](l: List[A]): Int
```

Additionally, throughout the paper we employ the following syntactic sugar for denoting lists.

```
[x1, ..., xn] ≡ Cons(x1, Cons(..., Cons(xn, Nil())...))
```

To implement a variational list, the first step is to introduce variation into the type signatures of the plain list operations.

[4] The `Option` type represents either a value as `Some(x)`, or no value as `None`. It is the Scala equivalent of the Haskell `Maybe` data type.

Already this poses a design decision: Which (argument and result) types do we make variational? This will determine where we introduce variation into the data structure. Then, we must decide how to encode this variation to support efficient implementations of the operations that we need.

### 4.1.1 `List[V[A]]`

An initial idea is to not change the types of the signatures or the list data structure at all. We have already seen that we can trivially implement variational lists (inefficiently) by applying `V` to `List[A]` to form a variational list of type `V[List[A]]`. However, we can also apply `V` to the element type `A`, to obtain a variational list of type `List[V[A]]`; that is, the variational list is just an ordinary list of variational elements. Now we can concisely represent the variation between `[1,2,3,4]` and `[1,2,5,4]` as `[1,2,T<3,5>,4]`.

The advantages of this implementation are that it supports the sharing of common elements among list variants (observe that there is no redundancy in the encoding of our example), and it requires no changes to the underlying list data structure. However, a significant drawback is that it enforces that all variant lists have the same length. For example, using this representation there is no way to encode the variation between the lists `[1,2]` an `[1,2,3]`. Therefore, while `List[V[A]]` is more space efficient, the naive `V[List[A]]` representation is more expressive. The next two implementations will recover this expressiveness while maintaining some or all of the space efficiency gains, but at the cost of increased implementation and runtime complexity.

### 4.1.2 `TList[A]`

Returning to the question of where to introduce variability in the types and implementation of our linked-list data structure, one possibility is to make the list's tail variational.

```
trait TList[A]
case class Nil[A]()                        extends TList[A]
case class Cons[A](h:A, t:V[TList[A]]) extends TList[A]
```

In addition to replacing `List[A]` by the new type `TList[A]`, observe that we have made the tail component of `Cons` variational. With this encoding, we can prepend an element onto potentially several alternative lists, supporting variational lists with different lengths that may share common prefixes. For example, we can encode the variation between `[1,2,3]` and `[1,2]` as `Cons(1, Cons(2, T<Cons(3,Nil()), Nil()>))`.

Implementations of the `get` and `length` operations are straightforward and exploit sharing at the head of the list.

```
def get[A](l: TList[A], i: Int): V[Option[A]] =
  l match {
    case Nil()     => One(None)
    case Cons(h,t) => if (i==0) One(Some(h))
      else flatMap(t, (k:TList[A]) => get(k,i-1)) }

def length[A](l: TList[A]): V[Int] = l match {
    case Nil()     => One(0)
    case Cons(h,t) =>
      map(flatMap(t,length[A]), (a:Int) => (a+1)) }
```

Note that the types of the operations also change to reflect the structure of the data type. The `length` operation now returns a variational integer since a `TList` represents potentially many different plain lists, which may be of different lengths. The `get` operation returns a `V[Option[A]]`. The outer `V` reflects that different variant lists can have different elements at index `i`. The `Option[A]` type reflects that `i` might be out of range for some of these variants, in which case the corresponding element in the variational result will be `None`.

This representation supports quickly adding new non-variational elements to a variational list (by the `Cons` operation). Since the first element of a `TList` is not varied, we must use `V[TList[A]]` if we want to support variation among arbitrary lists. The major limitation of this representation is that it supports only the efficient sharing of list prefixes. For example, to encode the variation between `[1,2,3,4]` and `[1,2,4]`, we must write the following, where the element `4` is repeated.

```
Cons(1, Cons(2, T<Cons(3, Cons(4,Nil())), Cons(4,Nil())>))
```

### 4.1.3 `OList[A]`

An alternative encoding of variational lists can be obtained by making the head of the list variational. In fact, we already explored this possibility in Section 4.1.1. The problem was that, by only varying the head, we were limited to representing variation among lists of the same length. To circumvent that problem in this implementation, we make list elements optional; that is, the head of a list may be a choice in which some of the variants are values and some are `None`.

```
trait OList[A]
case class Nil[A]() extends OList[A]
case class Cons[A](h: V[Option[A]], t: OList[A])
  extends OList[A]
def get[A](l: OList[A], i: Int): V[Option[A]]
def length[A](l: OList[A]): V[Int]
```

Besides replacing `List` by `OList`, the significant change is replacing the element type `A` by `V[Option[A]]` in both the argument to `Cons` and the result of `get`.

Using this data structure, we can represent the variational list `T<[1,2,3],[2,4]>` without redundancy as follows.

```
[Chc(T,One(Some(1)),One(None)), One(Some(2)),
 Chc(T,One(Some(3)),One(Some(4)))]
```

Observe that the second alternative in the choice containing `1` is `None`, indicating that there is no corresponding element in the list when tag `T` is not selected.

This encoding supports lists of different lengths (by using `None`), and sharing at arbitrary positions within the list, as demonstrated by the sharing of `2` in the middle of the example above. Therefore, it can efficiently represent all of the examples discussed in this section. Its main drawback is that it requires much more complex implementations of the operations `get` and `length`, since we must track which branches of variation ultimately contain elements and which do not.

### 4.2 Variational Maps

As with variational lists, the goal of a variational map is essentially to provide an efficient interface to a set of related alternative plain map data structures. Similarly to lists, the naive encoding `V[Map[A,B]]` is obviously inefficient since variational maps can be expected to share many entries.

One possible encoding of variational maps is `Map[A,V[B]]`. This representation exhibits many of the same tradeoffs as the variational list representation `List[V[A]]`; that is, its main advantage is that it can directly reuse an existing map data structure unchanged. Its main disadvantage is that it does not account for the possibility that different variant maps may contain entries for different sets of keys. However, for some application domains (where keys are fixed but values may vary), this representation may be a good choice.

For other cases, let us consider an approach similar to the variational list representation `OList[A]`. Specifically, we store mappings from keys to variational optional values. This is implemented by the following partial class definition.

```
class VMap[A,B](entries: Map[A,V[Option[B]]]) {
  def contains(key: A): V[Boolean] = ...
  def get(key: A): V[Option[B]] = ...
  def put(key: A, value: V[Option[B]]): VMap[A,B] = ...
}
```

The role of each operation is to translate an operation on `VMap` into an operation on the internal representation of `entries`.

As an extension, we illustrate below a general technique for incrementally building variational maps in a similar way as formula trees (see Section 3.2.2). This alternative implementation of `put` maps a key to a single value of type `B` in a particular variational context represented by a formula.[5]

```
  def put(key: A, ctx: Formula, v: B): VMap[A,B] =
    new VMap(entries + (key ->
            Chc(ctx, One(Some(v)), this.get(key))))
```

This `put` operation associates the key with a choice between either the new value or the value previously associated with that key. This allows us to build up a variational map entry piece-by-piece, rather than computing up front all alternative values that a key may map to. This is useful, for example, when accumulating data over another variational data structure. An obvious optimization to this implementation is to detect and remove dead branches from the choice structure using `prune` from Section 3.3.

### 4.3 Variational Sets

The last variational data structure we consider is a variational set. Once again, we use a strategy of pushing the `V` constructor into the definition of the data structure to increase sharing relative to the naive implementation of `V[Set[A]]`.

One possible implementation is `Set[V[Option[A]]]`, which is similar to `OList[A]` and the internal representation of `VMap[A,B]`. This implementation has the advantage of reusing an existing set implementation and it is maximally expressive.

An alternative implementation of variational sets is based on a map that associates each element with a formula. The formula defines in which configurations the element is present in the set. The motivation for this implementation is similar to the extended `put` operation for variational maps—it allows us to build up variational sets incrementally, rather than requiring us to compute in advance all configurations where an element is or is not present.

```
class VSet[A](entries: Map[A,Formula]) {
  def contains(key: A): V[Boolean] = ...
  def add(key: A, ctx: Formula): VSet[A] = ...
}
```

The internal representation is similar to the formula map implementation of `V`, described in Section 3.2.3, but simpler since it need not maintain the choice-as-partition invariant. This makes the `add` operation quite simple.

```
  def add(key: A, ctx: Formula): VSet[A] =
    new VSet(entries + (key ->
            ctx ∨ entries.getOrElse(key, False)))
```

When an element–formula pair is added, the element will be included in all configurations where either the argument formula is satisfied, or the previous formula for the element is satisfied. This supports incrementally adding new variational elements to the set based on the current configuration context.

## 5. Computing with Variational Data

Ultimately, variational data structures are needed to support variational computations. As described in Section 2.2, we focus specifically on functions that *preserve* the variability of their inputs in their output. Given a function `f` of type `(A1,...,An) => B`, the corresponding variation-preserving function `vf` has the type `(V[A1],...,V[An]) => V[B]`. Most importantly, the relationship between inputs and outputs defined by `f` will be preserved across all configurations in `vf`; that is, if we configure each variational input to `vf` with the same configuration `c` and obtain the plain inputs `a1`, ..., `an`, then if we also configure the variational output of `vf` with `c`, we should obtain the plain output `f(a1,...,an)`.

One important observation is that variation-preserving functions can be mechanically obtained from plain functions. As a simple example, consider the following function that adds two integers and returns the result.

```
def plus(a: Int, b: Int): Int = a + b
```

Our goal is to define a variation-preserving function `vplus` such that, for example, `vplus(A<1,2>,A<4,8>)` returns `A<5,10>` and `vplus(A<1,2>,B<4,8>)` returns `A<B<5,9>,B<6,10>>`. This can be easily achieved for any of the implementations of `V[A]` presented in Section 3 by the `map` and `flatMap` functions.

```
def vplus(va: V[Int], vb: V[Int]): V[Int]
  = flatMap(va, (a:Int) => map(vb, (b:Int) => a+b))
```

[5] The Scala syntax `map + (key -> value)` means to extend `map` with a new entry associating `key` to `value`.

In fact, we can lift *any* plain function to make it variation-preserving by using `map` and `flatMap`. The following functions automate this process for functions of different arities.[6]

```
def liftV[A,B](f: A=>B, va: V[A]): V[B]
  = map(va, (a:A) => f(a))

def liftV2[A,B,C](f: (A,B)=>C,
                  va: V[A], vb: V[B]): V[C]
  = flatMap(va, (a:A) => map(vb, (b:B) => f(a,b)))

def liftV3[A,B,C,D](f: (A,B,C)=>D,
                    va: V[A], vb: V[B], vc: V[C]): V[D]
  = flatMap(va, (a:A) => flatMap(vb,
                (b:B) => map(vc, (c:C) => f(a,b,c))))
```

Now we can define `vplus` as simply `liftV2(plus,va,vb)`.

While general, the `liftV` functions essentially execute the lifted function repeatedly on the cross product of the variants of its inputs, so this approach can be quite inefficient. Note also that we cannot generically lift operations on plain data structures to the variational data structures defined in Section 4 since these specialized implementations do not have types of the form `V[...]`. However, many operations can be defined by similar uses of `map` and `flatMap`. For example, the following function sums the elements of a variational `TList`, producing a variational integer as a result.

```
def vsum(l: TList[Int]): V[Int] = l match {
  case Nil()     => One(0)
  case Cons(h,t) => vplus(One(h), flatMap(t,vsum))
}
```

Implementing variational sum for `OList` requires also taking into account that some values may be `None`. These are counted as `0` when computing the sum.

```
def vsum(l: OList[Int]): V[Int] = l match {
  case Nil()     => One(0)
  case Cons(h,t) => vplus(
    map(h, (e:Option[Int]) => e.getOrElse(0)), vsum(t))
}
```

Although pattern matching, `map`, and `flatMap` provide a powerful interface for defining new operations on variational data structures, an important avenue of future research will be identifying more abstract interfaces that hide the implementation details discussed in Section 4. Ideally, clients need understand only the expressiveness and performance tradeoffs among representations and not their specific encodings.

While this section has presented a structured approach to defining simple variation-preserving functions, computing with variational data in real applications can get considerably more complicated. In the next section, we explore this problem and discuss how a suite of general-purpose, well-understood variational data structures can help.

## 6. Retrospective Design Analysis

In this work, we advocate a general approach to variability and start exploring the design space for variational data structures. In fact, we discovered only after several years of research that we and others have independently explored principles for encoding variational data. Without being aware of the larger design space, we have explored these encodings in an ad hoc fashion and incrementally improved them until they met some performance goals (for example, being able to parse the entire Linux kernel in reasonable time). While the different approaches are similar at a high level, they often use different encodings and make different tradeoffs, locating them at different points in the design space.

In this section, we retrospectively analyze the design of several systems that cope with variability in different ways using case-study research. We identify design decisions and their rationales at the time, as well as possible alternatives that our discussion of the design space reveals. We look mostly at systems that have been developed by some of this paper's authors, giving us access to the history and rationales of these projects. Since most of the systems rely on several years of research, it would require significant engineering effort to actually rewrite the systems to use more generic representations and to experiment with alternatives. Such an elaborate analysis exceeds the scope of this paper.

### 6.1 CIDE and CFJ

In our (Kästner and Apel) early work, we use a restricted encoding of variability based on optionality. CIDE is a tool for managing software variation by coloring parts of the code that correspond to different features [29]. Colored Featherweight Java (CFJ) [30] is a formalization of this technique based on Featherweight Java (FJ) [27], a formal calculus that models a small subset of the Java programming language. In both CIDE and CFJ, nodes in an AST can be marked as optional, but alternatives are not supported. Optional nodes are associated with formulas representing a partial configuration space.

The model is similar to the `OList` data structure introduced in Section 4.1. However, unlike list elements, not all AST nodes can be marked optional because omitting them would yield a syntactically invalid program. For example, a Java statement is syntactically optional while the condition of an if-statement is not. Therefore, to vary the condition of an if-statement, some workaround is needed, such as duplicating the entire if-statement, marking both as optional, and associating them with different (mutually exclusive) features. CIDE includes special support for some common patterns like this, but the problem could have been avoided with a variation model that also supports choices instead of only optionality.

### 6.2 Variability-Aware Type Checking in FFJ$_{PL}$

Our work on type checking FFJ$_{PL}$ is one of our earliest efforts to reason about variational programs [2]. FFJ$_{PL}$ extends FJ with support for feature composition based on mixins [24] and superimposition [3]. Each program feature is implemented by its own module; modules can be composed in different combinations, giving rise to a product line [3].

[6] These are analogous to Haskell's `liftM` functions for lifting a plain function into a monad.

The type system for $\text{FFJ}_{PL}$ must take the variability induced by the combinatorics of feature composition into account. The type of a term may vary depending on the features that are included. A key design decision was to represent the variational type of a term as the set of all possible types the term can have in the configuration space. Once this set has been computed for a term, the type checker proceeds with all possibilities simultaneously. Since the set contains only distinct types, this avoids redundant work for variant terms of the same type. This is an early example of exploiting sharing in a variational analysis.

Compared to later variational analyses, however, the variational data structures in $\text{FFJ}_{PL}$ are very simple—variational types are expressed as simple untagged sets of alternatives. This has a few limitations.

One limitation is that information, about which variants have which types, is lost during the typing process. For example, if method `m` has type `Int => Bool` when feature `T` is included, and type `Bool => Bool` otherwise, then the set of possible types is {`Int => Bool`, `Bool => Bool`}. Now every invocation of `m` must be compatible with both types, which in this case is impossible. Using a choice, we can represent the possible types of `m` as `T<Int => Bool, Bool => Bool>`. Then, in contexts where feature `T` must be selected, such as in the module implementing `T`, we need only check against the corresponding type of `m`. In contrast, the set representation leads to an overly conservative type system since it enforces that every use is compatible with every definition, even though some definition-use combinations are impossible.

Another limitation is that the set representation is flat, which misses opportunities for sharing in compound types. For example, the signature of method `m` varies only in its argument type, but we repeat the shared result type in both alternatives. The variational type of the corresponding choice representation is `V[A => B]`. A more efficient representation is to localize variability in compound types. For example, the type of `m` can be represented as `T<Int, Bool> => Bool`. This representation of variational types is used in the variational type inference algorithm described in Section 6.7.

At the time of developing $\text{FFJ}_{PL}$ and its type system, we did not have a full understanding of variational data structures and algorithms. The model we develop in this paper makes these limitations and their solutions apparent.

### 6.3 Variability-Aware Parsing in TypeChef

In the development of TypeChef, when we decided to reach for the goal of parsing the Linux kernel with all of its `#ifdef` configurations [31], we needed an efficient encoding of variability in the resulting AST. The size of the problem and the nature of having small `#ifdef` blocks in large files required a data structure where variation in the input (C source file) is represented locally in the output (AST).

After in-lining all included header files, a typical C file contains hundreds of top-level declarations (TLDs), many of which are guarded by `#ifdef` blocks. Some TLDs also have alternatives, for example, selecting between a 32-bit and 64-bit architecture. The average Linux file is affected by 207 configuration options [31], so a naive encoding, such as `V[AST]`, would lead to an exponential blowup. Instead, our data structure for a translation unit contains a list of TLDs (the order of TLDs is significant in C), where each TLD is associated with a formula describing in which configurations it is visible. The formula is derived from the condition of the `#ifdef` guarding the TLD, if any. This is equivalent to the variational list data structure `OList[TLD]`.

In retrospect, the chosen representation of `OList[TLD]` is better than `List[V[TLD]]` since it is common for different configurations to have different numbers of top-level declarations. It is also more efficient than `TList[TLD]` since many differences and commonalities are distributed throughout the entire list (in a C file, there are optional entries in header files at the beginning, and in the actual code at the end), which would lead to redundancies in `TList[TLD]`.

Deriving partial configurations from `#ifdef` expressions (where users can write `#if A || (B<2)`) requires an expressive representation of formulas. We use propositional formulas instead of also encoding numeric operations because reasoning about propositional formulas with SAT solvers is much faster than reasoning with CSP or SMT solvers.

Additionally, a TLD may itself contain variation. Where possible, we represent this by nested variational lists, for example, a variational list of statements, `OList[Stmt]`, in the block of a function. Where nodes of the AST are not syntactically optional, for example, the return type of a function or the condition of an if-statement, we encode variation by simple choices between alternatives. For choices, we use the formula tree representation, described in Section 3.2.2.

In the development of their SuperC parser [25], Gazzillo and Grimm independently arrived at the same data structure design as the TypeChef parser, except in an untyped setting.

### 6.4 Variability-Aware Type Checking in TypeChef

Using as input the variational AST data structure from Section 6.3, we implemented a type system that reports type errors in partial configuration spaces as output [32, 36]. The type checker works mostly in the usual way: It iterates over the AST, collects defined names and their types in a symbol table, and checks whether expressions are well typed. The major difference is that variation in the AST propagates to many other data structures. This has led to interesting observations regarding variational data structures.

We implemented a variational symbol table as a map from names to variational types, `Map[Name,V[Option[Type]]]`. As in the `VMap` data structure from Section 4.2, values are optional since not all symbols occur in every configuration. We had to develop most of the operations for this data structure from scratch, but would have used a general-purpose variational map data structure, such as `VMap`, if it was available.

In most cases, we expect relatively little variation within types, so we represent variational types as simply `V[Type]`.

This trades the ability to share sub-parts of types for simplicity. Some exceptions are (anonymous) structure and union types, which may have many fields, often with optional or alternative entries. Therefore, for structure and union types, we represent variational fields using the same variational map data structure developed for the symbol table.

In the AST, we represent variational lists of type specifiers (e.g. **`const`**, **`short`**, **`int`**) as `OList[Specifier]`. When deriving a variational type, we first expand the variational list into a choice of plain lists, `V[List[Specifier]]`. Then, we map a plain `getType` function, `List[Specifier] => Type`, across the choice to produce a variational type, `V[Type]`. This approach does not exploit sharing in the variational specifier list, but in practice, we expect optional specifier lists to be relatively short, so there is a low combinatorial explosion.

### 6.5 Choice Calculus

In a different line of work, we (Erwig and Walkingshaw) have introduced the *choice calculus* as a formal model of variation that can be instantiated for different object languages and extended by new language features [20, 21]. The goal of the choice calculus is to provide a platform for research on the representation, manipulation, and analysis of variation. In fact, we have used it in this role throughout this paper, as it is the basis of the choice-based variation models from Section 3.

The choice calculus is similar to the tag-tree model described in Section 3.2.1. The main difference is that each choice is locally bound by a *dimension* of variation that declares several tags to be used by all of its bound choices. All choices in the same dimension must have the same number of alternatives, and they are all synchronized; that is, if the first tag in a dimension is selected, every bound choice is replaced by its first alternative.

The tag-tree model is isomorphic to the choice calculus restricted to binary dimensions and choices. An $n$-ary choice can be transformed into a tag tree by simply chaining binary choices, for example, if dimension `D` contains tags `T`, `U`, and `V`, then the choice calculus expression `D<1,2,3>` can be encoded by the tag tree `T<1,U<2,3>>` (the selection of `V` implies that neither `T` nor `U` was selected). Therefore, although tag trees are equally expressive as the choice calculus, the organization of tags into dimensions imposes an additional structure that ensures that mutually exclusive tags are used consistently.

Otherwise, relative to formula-based representations, such as formula trees and TypeChef, the choice calculus exhibits similar tradeoffs as tag trees: Some kinds of variation are difficult to represent efficiently, for example, inclusive-or relationships between tags can only be expressed by redundancy or by a separate reuse construct provided by the choice calculus (similar to a let-expression). However, it supports simple functional operations in many places where formula-based representations require SAT solvers. Also, like tag trees and formula trees (but unlike formula maps), the choice calculus preserves the choice-as-partition invariant by construction.

### 6.6 Variation Programming

In our work on *variation programming* [22], we promoted the idea of computing with variation and began exploring variational data structures, using the choice calculus. We provided a general strategy to add variability to an algebraic data type and to lift operations on the original data type to the new variational data type.

The strategy is to extend the data type `T` by a new case for `V[T]` for representing variation embedded within the data type. Since `V` is a monad, the `map` and `flatMap` functions functions can be mechanically derived, along with the `liftV` functions from Section 5. This makes it trivial to lift functions on `T` to the new data type. In the case of lists, this approach corresponds exactly to the `TList[A]` representation for incorporating variability in the tail of a list.

This systematic approach to defining variational data structures is general in that it can be applied to any data structure that can be represented as an algebraic data type. However, it only supports the sharing of contexts of variational subexpressions; that is, if there are more commonalities within the alternatives of a choice, they cannot be shared. Therefore, for many use cases a specialized representation will be more efficient; for example, this weakness of the mechanically derivable `TList[A]` is overcome by `OList[A]`.

### 6.7 Variational Type Inference

Using the choice calculus, we have also worked on the problem of typing variational programs, but from a quite different perspective than $\text{FFJ}_{PL}$ and TypeChef. We have extended Hindley-Milner-style type inference to the variational lambda calculus (VLC) [13, 14]. The type system assigns correspondingly variational types to VLC expressions.

Both VLC expressions and variational types can be viewed as variational data types of the form described in Section 6.6. The representation of variational types is more space efficient than $\text{FFJ}_{PL}$ and TypeChef since it supports sharing of common subparts of a type. This can have a significant effect when several dimensions of variability are involved. For example, the following variational type includes three dimensions of variation.

```
(A<Int,Bool>, B<Int,Bool>) => C<Int,Bool>
```

In TypeChef, a function with this type would require eight alternative types in the formula tree corresponding to the function's declaration. This is acceptable since such types are rare in existing code, which is TypeChef's focus. However, we sought to promote variability to a first-class language feature and encourage its use, so efficiently representing highly variational types was a priority.

The efficient representation of variation in VLC expressions and types is crucial to the efficiency of our type-inference algorithm. For expressions, localizing variation in choices allows us to infer the types of shared context once for all variants. It also allows us to locally reuse type unification results when two subexpressions have the same type. For types, localized variation supports a type unification algorithm that is cubic in the size of the types being unified. While this is somewhat worse than traditional type unification, it is significant that the running time is bounded by the size of the expressions rather than the number of variants, which is the typical source of blowup when analyzing variability.

### 6.8 Variability-Aware Data-flow Analysis

In SPL$^{\text{LIFT}}$, we (Bodden) have presented an approach to automatically lift inter-procedural data-flow analyses to operate on an entire software product line at once [9]. The main design goal of SPL$^{\text{LIFT}}$ was to support the direct reuse of single-program data-flow analyses on variational programs, and this is reflected in the design of its data structures.

The analysis operates on a variational inter-procedural control-flow graph in which individual edges are annotated with Boolean feature constraints. The encoding of successors is effectively a variational map between statements, `VMap[Stmt,Stmt]`, which conditionally connects one statement to another through a control-flow edge. This encoding is a natural extension of the edge map representation for plain graphs. For control-flow graphs, which are typically quite sparse, the encoding is efficient. A viable alternative implementation would be to maintain a variational list of successor nodes for each node in the graph, for example, as `OList[Stmt]`. Two nonviable alternatives are `V[List[Stmt]]`, which does not exploit sharing, and `List[V[Stmt]]`, which requires each node to have the same number of successors in every configuration. It is interesting to note that the designers of TypeChef independently arrived at the same variational map-based encoding of variational control-flow graphs [36].

While executing an analysis, SPL$^{\text{LIFT}}$ associates computed data-flow facts with formulas that describe the configuration space in which the fact is known to hold so far. This association is implemented by a variational set of data-flow facts, `VSet[D]`. This encoding was chosen to integrate easily with the existing analysis engine for plain Java programs. SPL$^{\text{LIFT}}$ is a variant of the IFDS framework for *interprocedural finite distributive subset* problems [41]. Within this framework, one can safely process a single data-flow fact at a time. This means that, for plain programs, at each statement the analysis would iterate over the set of known facts, `Set[D]`, and apply its data-flow function to each one. The lifted implementation similarly processes the variational set, considering each fact and its corresponding formula. This makes the lifting process completely transparent from the perspective of the existing data-flow analysis.

Brabrand et al. [11] also discuss a number of different encodings to lift data-flow analyses from regular programs to software product lines. In particular, they discuss two different variational encodings of data-flow facts, which they call A3 and A4. Both effectively implement instances of variational sets, A3 as an equivalent of `V[Set[A]]`, and A4 as a variant of `VSet[A]`.

## 7. Related Work

The idea for this paper formed at the Dagstuhl seminar on "Analysis, Test and Verification in the Presence of Variability" [10], where we discussed the need for foundational work on variational data structures. There had already been some preliminary attempts to identify general principles for variational data structures and algorithms: In the context of variational testing and model checking, respectively, Kästner et al. [33] and Apel et al. [5] identified the principles of *late splitting* and *early joining* as essential for efficiently computing with variability. In the context of type checking and data-flow analysis, Liebig et al. [36] emphasized the need to express variation locally within data structures. Also in the context of data-flow analysis, Brabrand et al. [11] explored different ways to express and reason about variability at the level of data structures. While none of these findings were generalized beyond their respective contexts, they motivated us to put variational data structures on more solid ground.

In a more foundational line of work, Erwig et al. developed an abstract representation of variational sets and graphs, and a framework for describing variational graph algorithms [23]. The representation is based on tagging the components of the data structures with Boolean inclusion conditions.

The rest of this section focuses on the substantial corpus of work on modeling and reasoning about variation in specific application areas. In this work, much like with our own early attempts (see Section 6), variation is often encoded implicitly or in an ad hoc fashion. We discuss only a representative subset; for a comprehensive overview on variational program analyses, we refer the reader to the survey by Thüm et al. [43].

Early work on analyzing variational programs used different strategies to represent and reason about variability. The seminal work of Czarnecki et al. [18] and Thaker et al. [42] encoded well-formedness and type-safety properties of configurable models and programs as SAT problems. The key idea was to construct a single Boolean formula that captures the whole compile-time variability of a model or program, then to verify whether it is consistent with a variability model that expresses the intended variability. Although there were no variational data structures involved, the idea of sharing to speed up computations with variation was already at the heart of this work. Our later work on type checking made variation more explicit, as explained in Section 6.

Work on variational model checking tries to maximize sharing when representing and analyzing the states of all program variants [4, 15, 16, 19, 35, 40]; that is, parts of the state space that are equal across multiple variants should be explored and analyzed only once. To achieve this goal, one can either map parts of the state space to partial configurations [16, 35] or encode the association of program states and configuration options in dedicated configuration-option variables [4, 15, 40]. Either way, the model checker must find a compact representation of the state space for all program variants, and state-space exploration must reason about sets of states that correspond to partial configuration spaces.

In work on testing, *combinatorial testing* addresses the exponential blowup of configurations by strategically sampling the configuration space and applying traditional testing methods to a smaller set of variants [17, 38]. Toward efficiently testing *all* variants, several researchers have explored executing a program on variational inputs by lifting a corresponding interpreter [33, 34, 37, 45]. Variational interpreters have also been employed for computing with alternative privacy policies [6, 7, 46]. Values in the store of a variational interpreter are represented by variational data types. These approaches use different variational data structures; for example, Austin, Yang, and their collaborators have essentially designed tag trees [6, 7, 46], whereas Kästner et al. have used formula trees and variational maps [33].

## 8. Conclusion

Variation is a considerable source of complexity in software systems. Despite many attempts to represent and reason about variation in software, there is no principled understanding of how to manage variational data effectively, nor of the design space and implementation tradeoffs of variational data structures. So far, researchers and practitioners have mostly resorted to ad hoc solutions, which are not easily generalizable to other use cases and therefore miss opportunities for knowledge and code reuse. Moreover, because there are many other potential applications of variational data structures, it misses a chance for research on software variability to make a more general and significant contribution to software engineering at large.

The main goal of this paper is to promote systematic and foundational research on variational data structures and to raise awareness of the benefits of such research. Our key insight is that support for variation can be understood as a general and orthogonal property of data types, data structures, and algorithms. We began the systematic exploration of some basic variational data structures, focusing on revealing tradeoffs among different implementations. Based on this work, we retrospectively analyzed the design decisions and design tradeoffs in our own previous work and in the work of other researchers. However, this is only the beginning. This paper is also a call to action to rethink current approaches to managing variational data, to examine how they can be generalized, and to develop new solutions based on a principled and common understanding of the nature of variation.

## Acknowledgments

This work is supported by NSF grants CCF-1219165, CCF-1318808, and IIS-1314384; by DFG grants AP 206/2, AP 206/4, and AP 206/6; by the BMBF within EC SPRIDE; and by the Hessian LOEWE excellence initiative within CASED. We would also like to thank Paolo G. Giarrusso for helpful comments on an earlier version of this paper.

## References

[1] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Trans. Programming Languages and Systems*, 32(5):1–33, 2010.

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

[3] S. Apel, C. Kästner, and C. Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Trans. Software Engineering*, 39(1):63–79, 2013.

[4] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

[5] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: Case studies and experiments. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. Int'l Symp. Principles of Programming Languages (POPL)*, pages 165–178. ACM, 2012.

[7] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proc. ACM SIGPLAN Work. on Programming Languages and Analysis for Security*, pages 15–26. ACM, 2013.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Engineering*, 30(6):355–371, 2004.

[9] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPL$^{\text{LIFT}}$: Statically analyzing software product lines in minutes instead of years. In *Proc. Int'l Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM, 2013.

[10] P. Borba, M. B. Cohen, A. Legay, and A. Wąsowski. Analysis, test and verification in the presence of variability (Dagstuhl seminar 13091). *Dagstuhl Reports*, 3(2):144–170, 2013.

[11] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM, 2012.

[12] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *Proc. Int'l Symp. Principles of Programming Languages (POPL)*, pages 583–594. ACM, 2014.

[13] S. Chen, M. Erwig, and E. Walkingshaw. An error-tolerant type system for variational lambda calculus. In *Proc. Int'l Conf. Functional Programming (ICFP)*, pages 29–40. ACM, 2012.

[14] S. Chen, M. Erwig, and E. Walkingshaw. Extending type inference to variational programs. *ACM Trans. Programming Languages and Systems*, 36(1):1:1–1:54, 2014.

[15] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 321–330. ACM, 2011.

[16] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l*

*Conf. Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

[17] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA)*, pages 129–139. ACM, 2007.

[18] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM, 2006.

[19] M. d'Amorim, S. Lauterburg, and D. Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *Proc. Int'l Symp. on Software Testing and Analysis*, pages 50–60. ACM, 2007.

[20] M. Erwig, K. Ostermann, T. Rendel, and E. Walkingshaw. Adding configuration to the choice calculus. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 13:1–13:8. ACM, 2013.

[21] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

[22] M. Erwig and E. Walkingshaw. Variation programming with the choice calculus. In *Generative and Transformational Techniques in Software Engineering IV*, LNCS 7680, pages 55–100. Springer, 2013.

[23] M. Erwig, E. Walkingshaw, and S. Chen. An abstract representation of variational graphs. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2013.

[24] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. Int'l Symp. Principles of Programming Languages (POPL)*, pages 171–183. ACM, 1998.

[25] P. Gazzillo and R. Grimm. SuperC: Parsing all of C by taming the preprocessor. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 323–334. ACM, 2012.

[26] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *J. Object Technology*, 7(3):125–15, 2008.

[27] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Programming Languages and Systems*, 23(3):396–450, 2001.

[28] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Pittsburgh, PA, 1990.

[29] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM, 2008.

[30] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Software Engineering and Methodology*, 21(3):14:1–14:39, 2012.

[31] 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 *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[32] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792. ACM, 2012.

[33] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward variability-aware testing. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2012.

[34] C. H. P. Kim, S. Khurshid, and D. Batory. Shared execution for efficiently testing product lines. In *Proc. Int'l Symp. Software Reliability Engineering (ISSRE)*, pages 221–230. IEEE, 2012.

[35] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

[36] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable analysis of variable software. In *Proc. Europ. Software Engineering Conference and ACM SIGSOFT Symp. Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013.

[37] H. V. Nguyen, C. Kästner, and T. N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 907–918. ACM, 2014.

[38] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11:1–11:29, 2011.

[39] W. Osler. On the educational value of the medical society. *Yale Medical Journal*, IX(10):325, 1903.

[40] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE, 2008.

[41] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. Int'l Symp. Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995.

[42] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

[43] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 2014. To appear.

[44] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based theorem proving for deductive verification of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM, 2012.

[45] J. Tucek, W. Xiong, and Y. Zhou. Efficient online validation with delta execution. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 193–204. ACM, 2009.

[46] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proc. Int'l Symp. Principles of Programming Languages (POPL)*, pages 85–96. ACM, 2012.