Variational Lists: Comparisons and Design Guidelines*

Karl Smeltzer
Oregon State University
USA
smeltzek@oregonstate.edu

Martin Erwig
Oregon State University
USA
erwig@oregonstate.edu

Abstract

Variation is widespread in software artifacts (data and programs) and in some cases, such as software product lines, is widely studied. In order to analyze, transform, or otherwise manipulate such variational software artifacts, one needs a suitable data structure representation that incorporate variation. However, relatively little work has focused on what effective representations could be to support programming with variational data.

In this paper we explore how variational data can be represented and what the implications and requirements are for corresponding variational data structures. Due to the large design space, we begin by focusing on linked lists. We discuss different variational linked-list representations and their respective strengths and weaknesses. Based on our experience, we identify some general design principles and techniques that can help with the development of other variational data structures that are needed to make variational programming practical.

CCS Concepts • **Software and its engineering → Data types and structures**; *Software design tradeoffs*;

Keywords data structures, variation representation, language design

ACM Reference Format:
Karl Smeltzer and Martin Erwig. 2017. Variational Lists: Comparisons and Design Guidelines. In *Proceedings of 8th ACM SIGPLAN International Workshop on Feature-Oriented Software Development (FOSD'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3141848.3141852

*This work is supported by the National Science Foundation under the grant IIS-1314384.

FOSD'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5518-6/17/10…$15.00
https://doi.org/10.1145/3141848.3141852

1 Introduction

Variation in software and data is ubiquitous. Version control systems and software product lines are two wide-spread approaches to managing variation in software. Variational data does not just occur when viewing programs as data, however, but also in many other applications. This is because instead of executing the same program repeatedly on different inputs, one can envision running the program only once on one variational input. Examples include searching for a specific string in all variants of a version control system [19], in the history of an editor's undo stack [26], or in speculatively merged branches [5]. Performing such a search repeatedly in each variant is potentially very inefficient. Since different variants often have large parts in common, a more promising approach would be to represent all documents as one document with differences expressed locally, and then executing the search over this combined, variational document. This would help avoid unnecessary repeated searches over the same parts and could overall speed up the search considerably. To achieve this we need a representation for variational data in general as well as new variational operations and control structures.

In this paper we report on variational lists as a fundamental data structure, which can serve as a basis for many applications including the search over variational documents mentioned above. Conceptually, a variational list is a mapping from configurations to plain (non-variational) lists. A configuration is a structure that uniquely identifies a particular variant among all the variants in the variational list. This is best illustrated with a specific example. Consider this example code.

```
#ifdef WINDOWS
   printf("Hi Redmond.");
#elif defined UNIX
   printf("Hello");
   #ifdef MAC
       printf(" Cupertino.");
   #elif defined LINUX
       printf(" Helsinki.");
   #endif
#endif
printf("Goodbye.");
```

When source code is annotated with C preprocessor (CPP) macros for conditional compilation (`#ifdef` and family), we can view that code as a variational list of code sections, i.e., each the macros define configurations which determine a

particular list of sections and thus a particular variant of the code.[1]

Analyzing this code naively will need to generate and analyze all $2^4 = 16$ source code files individually, creating extra work. By first translating this code to a variational list where the elements store sections of the program text, and then performing a variational search on it, we stand to potentially avoid that repeated work.

We could define such a variational list with a mapping from configurations to their corresponding variants (uninteresting cases omitted):

$$c_0 = \{\} \qquad c_1 = \{\texttt{WINDOWS}\}$$
$$c_2 = \{\texttt{UNIX, MAC}\} \quad c_3 = \{\texttt{UNIX, LINUX}\}$$

$$\begin{aligned}
\{\ c_0 &\mapsto [\texttt{printf("Goodbye.");}],\\
c_1 &\mapsto [\texttt{printf("Hi Redmond.");, printf("Goodbye.");}],\\
c_2 &\mapsto [\texttt{printf("Hello");,printf(" Cupertino");,}\\
&\quad\ \texttt{printf("Goodbye.");}],\\
c_3 &\mapsto [\texttt{printf("Hello");printf(" Helsinki");,}\\
&\quad\ \texttt{printf("Goodbye.");}]\}
\end{aligned}$$

Each plain list used in this map is called a *variant* of the variational list, and the keys in the map amount to *decisions* that one has to make in order to access any particular variant. Each subset of macros corresponds to a potential configuration that can appear as a key in the mapping. Any representation of variational values consists in some way of these two components: (a) the individual variants, and (b) the decisions to distinguish between the variants.

We could also swap the usage of the list and map, so every list element is a map to plain values. This may seem inconsequential, but we will demonstrate small changes result in surprisingly different performance. This is hinted at by the duplication of `printf("Goodbye.");` in the example map. We also still need to figure out how programmers can actually work with these structures.

In the remainder of this paper we demonstrate several steps towards solving these problems. We begin by exploring the design space for variational list data structures. By examining how they support sharing and evaluating their performance, we identify what approaches are sufficiently robust, efficient, and generally useful for variational programming.

Specifically, we will discuss two different representations for variational values in Section 2. In Section 3 we will then describe how these representations can be employed in different ways to add variation to lists. In addition to the interaction of placement of variants and list position, the efficiency of different variational list representations also depends on the degree of sharing that a representation can achieve. We will therefore identify different forms of sharing in Section 4 and analyze what kind of sharing can be achieved by the different variational list representations.

Section 5 demonstrates some of the requirements and implications of realistic uses of these lists, and operations on them. In Section 6 we discuss performance implications of the representations abstractly, and then proceed into a set of representative benchmarks. Finally, we discuss related work in Section 7 and summarize what we have learned in Section 8.

This paper makes the following specific contributions (for clarity we mention include but annotate some contributions made in prior work).

- A systematic *structuring* of the design space for variational list data structures.
- A *comparison* of specific variational list representations.
- A *breakdown* of the kinds of sharing exhibited.
- Insights into the design of a variational list programming *interface*.
- A brief characterization of *runtime efficiency* for variational lists and a set of *benchmarks* comparing performance on real-world data.
- An *alternative interface* design to improve underperforming list representations.

2 Representing Variational Values

Before focusing on variational data structures in general and lists in particular, we need a systematic account of what variation is and what variational values are. Based on a sound formalism for variation we can build and compare techniques for variational data structures.

We discuss two major approaches to variation representation that differ fundamentally in the way they group and share non-variational data and in the way variational data is accessed, although other techniques are also possible. Section 2.1 presents an approach that forms the formal basis for a number of different representations, including the mapping representation shown in the Introduction. The employed view of variational data is that of plain data being dependent on decisions about which variants they belong to. In other words, decisions are mapped to plain data. In contrast, the approach illustrated in Section 2.2 maps plain data to decisions about which variants they belong to. Both representations can be used as an implementation for a generic variational type constructor `V`.

2.1 Named Choices and Choice Trees

The choice calculus [10] regards variation as named choices between alternatives. For example, the choice between the two plain (non-variational) values `1` and `2` is written as $A\langle 1, 2\rangle$. The plain values are called *alternatives* of the choice, and the name A is called its *dimension*. Each dimension A gives rise to two *selectors*, A and $\neg A$, that select from a choice tagged

[1] Variational data arises from a surprising number of places, for example, data from different sources about political polling, rankings of all sorts, weather data, and so on.

by that dimension the right and left alternative, respectively. When multiple choices exist in a single dimension, they are said to be *synchronized*, meaning that any time one of the alternatives from one choice is selected, the corresponding alternative in every other choice in that dimension must also be selected.

Choices can be nested inside one another, as in the expression $A\langle B\langle 1, 2\rangle, B\langle 3, 4\rangle\rangle$. These nested choices capture the situation in which we need to select an alternative (left or right) for both the A dimension and the B dimension in order to reach a plain value. In this work choices are always binary. That is not a limitation of the choice calculus, but rather a way of simplifying examples.

This variation representation has been called a *tag tree* in Walkingshaw et al. [24] (where the tags correspond to dimensions), but here we prefer the name *choice tree* to differentiate it more clearly from tag-based approaches, discussed below. Each leaf in a choice tree represents a variant, and any path from the root to a leaf represents a decision.

The use of maps for encoding the relationship between configuration/decisions and values (as done in the examples from the Introduction) can always be encoded with binary choices. For example, for a map $M = \{A \mapsto x, \ldots\}$ we can construct a binary choice $A\langle \ldots, x\rangle$, and then recursively add other nested choices in the left branch of the existing choice for every key-value pair in the map.

It may seem intuitive to use such a map directly instead of choices. Recall, however, that such a map will grow exponentially with the number of configuration options and is therefore impractical for real scenarios.

2.2 Tag Formulas and Tag Maps

Inverting the direction of access to variants in variational data provides an expressive and powerful way to maximize sharing in data (see also Section 4). In this approach, each plain value is tagged to indicate which variants include it. For example, the value 1^A indicates a value 1 which is included when tag (or dimension) A is selected, but not otherwise. Like the choice calculus, this relies on some labeling scheme in which a selection of labels forms a decision that identifies the desired variant. This approach can be extended to allow for tag formulas, analogous to boolean formulas. The advantages of this extension are twofold. First, identical plain values need not be duplicated. Instead of $[1^A, 1^B]$ we can simply write $[1^{A \vee B}]$. Second, it allows more complex relationships to be defined such as values which are only included when multiple selections are made. For example, $1^{A \wedge B}$ indicates that the plain value 1 is only included when both A and B are selected.

With tag formulas, a variational value can be implemented as a map from plain values to tag formulas. We call this representation a *tag map*.

The major downside to this approach is that some operations require the use of a SAT solver. Moreover, an expensive normalization process is sometimes needed to maintain sharing.

Both choice trees and tag maps can serve as underlying implementations for a variational type constructor `V` that can be used to represent arbitrary variational values. For example, the type `V Int` would represent variational integers and could correspond to either choices of integers or tagged integers. We distinguish between these two approaches, where necessary, by referring to them as `VChc` and `VTag`, respectively. In cases where the distinction is not necessary, we simply use `V`.

2.3 Visual Notation

In Section 3, we will make use of a compact visual notation when presenting examples. This notation is based on graphs whose nodes represent data and dimensions/tags and whose edges represent pointers in the data structure. In cases where both `VChc` and `VTag` can be used, we only show the structure of choices for brevity's sake. This notation, applied to the map of lists example from the Introduction, is shown in Figure 1.

Every binary choice is represented as a node with two outgoing edges corresponding to the choice's two alternatives. In general, rather than rendering this choice tree everywhere it occurs, we employ syntactic sugar which depicts the entire tree structure as a single, unlabeled node. The outgoing edges are labeled with configurations. We also add color to help distinguish particular variants. In this example, we would condense the graph to the one in Figure 2a.

3 Variational List Representations

We next explore the design of variational linked list data structures. While there is only one way to apply a type constructor to a value, there are several different approaches to do so with data structures. First, we can simply apply the type constructor at the *top level*. That is, just as we might have a `V Int` for a variational integer, we could also define a `V [Int]` for a variational list of integers. But we could also distribute the variational type constructor *parametrically* throughout the list, giving us a list of variational integers `[V Int]`. Finally, we could take a *recursive* approach, where rather than reusing the `V` constructor, the variation definition is woven directly into the recursive list definition.

By systematically combining the different implementations of `V` together with these approaches for integrating the variation into the list structure, we obtain six candidate variational list representations. An overview of how each of the following representations is categorized is shown in Table 1.

For each representation, we summarize the main idea and explain how it impacts the implementation of typical list functions. When we talk about a variational version of a common function, it should be understood to mean the function obtained by *variationalizing* all of the inputs

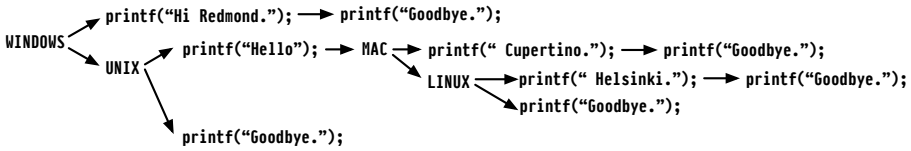
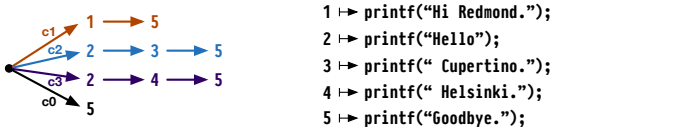
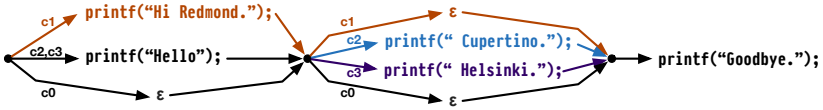


Figure 1. The unabbreviated visual notation used to demonstrate the general shapes that variational lists take.

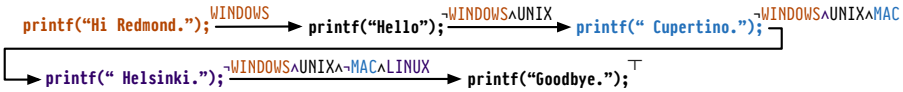
(a) Box list. Every variant list is stored in its entirety and nothing is shared.



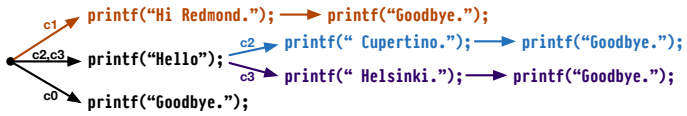
(b) Pointer list. Every element is stored in a hash table and only the indices are stored in the list.



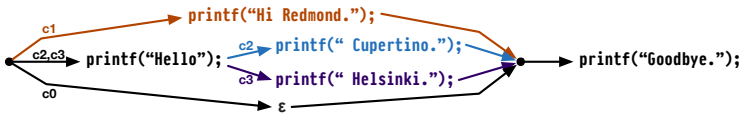
(c) Choice list



(d) Tag list



(e) Suffix list



(f) Segment list

Figure 2. Example variational lists containing conditional pre-processor annotated source code from the Introduction making use of the visual notation introduced in Section 2.3.

and outputs of that function. For example, where the plain version of `head` takes a plain list and produces a plain value, variational `head` takes a variational list as input and produces a variational value as output.

3.1 Box Lists

A straightforward variational list representation is obtained by applying the generic variational type constructor `V` to a list.

Since this approach does not have access to the individual variant lists and stores each of them as whole, we call these lists *box lists*. We use the generic `V` rather than `VChc` or `VTag` specifically, because either is adequate here.

```
type BList a = V [a]
```

See Figure 2a for an example box list.

This representation is essentially a brute force approach. This makes it easy to lift plain list operations using an implementation of a function `vMap` for the `V` type constructor that applies a function to all variants. Here, and going forward, the code we reproduce is based on our Haskell prototype but since the ideas presented generalize we have elected to take some simplifying syntactic liberties, including using choice syntax in place of actual constructors, which could make our definitions appear type incorrect to Haskell experts.

```
vMap :: (a -> b) -> V a -> V b
vMap f A⟨x, y⟩ = A⟨vMap f x, vMap f y⟩
vMap f x       = f x
```

The implementation of `vMap` for the variant map implementation from the Introduction is more involved since it has to potentially merge variants in the result that are mapped by `f` to the same value, see also Walkingshaw et al. [24].

With `vMap` we can define many lifted variational functions by just applying it to plain versions.

```
vHead :: BList a -> V a
vHead = vMap head
```

Since they are so straightforward, box lists serve as a base case for comparing the remaining representations.

3.2 Pointer Lists

We can extend the box list by a level of indirection, which allows common elements to be shared but at the cost of increased overhead. We refer to lists using this extension as *pointer lists*. We apply `V` to keys, which index into a map storing the actual list elements, ensuring no duplication.

```
type PList a = (Map Int a, V [Int])
```

An example pointer list is shown in Figure 2b.

The implementation of pointer list operations is similar to the box list case since it is a direct extension, except that we often need to compose them with a lookup operation which extracts values from the map.

```
vLookup :: V Int -> Map Int a -> V a

vHead :: PList a -> V a
vHead (m,vi) = vLookup (vMap head vi) m

vReverse :: PList a -> PList a
vReverse (m,vi) = (m, vReverse vi)
```

This approach adds overhead, especially since Haskell maps are implemented as trees, but enables sharing of all common elements in all variants. This complete sharing is unique in the representations considered here. Indirection could also be applied to any of the following representations.

Table 1. A categorization of variational list representations according to their use of variational type constructors and how they are integrated into the list structure.

		Use of `V`		
		Top Level	**Parametric**	**Recursive**
`V`	`VChc`	box, pointer	choice	suffix, segment
	`VTag`		tag	

3.3 Choice Lists

Since a list is itself a generic type constructor, we can move the `V` type constructor to its argument type and apply variability not on the level of whole lists, but on the level of elements. This gives us a single large list in which each element is itself a choice tree of variational list elements. Such an approach fails immediately, however, because variant lists of different lengths cannot be modeled. Choices contain values for all decisions, so there is no way to include a value which is only relevant to a subset of variant lists.

Our workaround is to allow certain choice tree leaves to be empty. That is, we modify the use of the `VChc` type constructor to carry values of type `Maybe a`, which allows us to represent partial variational values.

```
type CList a = [VChc (Maybe a)]
```

These are called choice lists and an example is shown in Figure 2c. As discussed in Section 4, there is generally no canonical choice list for a given set of data.

Any list operation which returns or manipulates single list elements will now require additional work. To best understand this, consider how to implement a `vHead` function for this representation. We might be tempted to write something like the following.

```
vHead :: CList a -> VChc (Maybe a)
vHead (x:_) = x
```

This is problematic because the first element could have empty leaf values even when the list variant is not empty. Consider the following variational list:

```
[A⟨Just 1, Nothing⟩, A⟨Just 2, Just 3⟩]
```

Neither variant is empty, but the first element of the choice list only contains a value relevant to one of the two. This means that the implementation for `vHead` given above produces `A⟨Just 1, Nothing⟩`, which is incorrect.

A correct implementation is more complicated and needs to potentially traverse the entire choice list to ensure that it finds a value (if one exists) for every variant. This results in a runtime complexity for `vHead` that is no longer constant time. Similar issues arise in other cases, too, such as sorting. Section 6 discusses this further.

3.4 Tag Lists

Tag lists use the same parametric application of `V` as choice lists, but with the tag formula implementation of `V` instead of choices. Tag formulas are paired with each element in a plain list, indicating which variants include those elements and which do not.

```
type TList a = [VTag a]  -- using the list as a map
```

See Figure 2d for a tag list example. These suffer from some of the same problems as choice lists, e.g., the implementation

of `vHead` is slow in both. Consider the list $[1^A, 2^A, 2^B, 1^B]$. In order to return the first element for both variants A and B we need to look beyond the first element, which has a negative impact on runtime complexity.

Tag lists have one major advantage, however, in that many plain list operations work as is. For example, the typical `reverse sort` functions work just fine. When the overall tag list is reversed or sorted, so is any particular variant.

Conversely, tag lists also have some drawbacks. Consider how we might implement a fold, for example. Since the tag map values we encounter as we traverse a list have no structure, we have to check the annotated tags and perform Boolean satisfiability computations to determine how to proceed.

This demonstrates that even simple, seemingly transparent changes to the way we build our variational data structures (in this case the implementation of `V` inside our list) can have far-reaching and non-intuitive effects, both good and bad.

3.5 Suffix Lists

As an alternative to using traditional linked lists as a base, variation can be integrated directly into the structure of the list. One such approach, first demonstrated in Erwig and Walkingshaw [11], is to extend traditional cons lists with a nesting constructor that uses `V`. We call these suffix lists.

```
data SuffList a = Cons a (SuffList a)
                | Nil
                | Nest (VChc (SuffList a))
```

See Figure 2e for a suffix list containing example data. This approach produces tree shapes rather than graphs, and once two variants have split apart, they will never be merged again.

Suffix lists share common elements among variants in the prefix of variational lists. From the first element up until the first point of variation, we have something indistinguishable from a traditional cons list. The actual sharing achieved depends on the data being stored in the lists, which makes it difficult to reason about in general. More details follow in Section 4.

As is becoming a theme, this representation encounters tradeoffs when implementing list operations. Generally, those which iterate through the list but do not modify the structure or reorder anything perform well on suffix lists. For example, `head` does not need to traverse the whole list, as was required for choice and tag lists, and functions like `last` are straightforward since there are no empty elements.

Conversely, functions that change the structure of the list such as `sort` and `reverse` are challenging. The latter is a clear example. Since suffix lists are actually trees, which have one root node and typically many leaf nodes, it is not possible to simply reverse the pointers. Instead, we need to find all of the leaf nodes and combine them into a root node, and then work backwards. Compared to the simple one-pass list reversal operations that work for other lists, this is inefficient.

3.6 Segment Lists

The idea of *segment lists* is to encode variational lists as a sequence of segments, where a segment is either a named choice of nested lists or a sequence of plain elements.

```
type SegList a = [SegListV a]
data SegListV a
  = Elems [a]
  | Split Name (SegList a) (SegList a)
```

See Figure 2f for an example segment list. The segment list is similar in principle to the choice list, except more compact. Because of this, it shares most the tradeoffs with the parametric application of variation in the choice lists: We generally need to traverse the entire list in order to implement `vHead` function, and we need to add extra machinery to manipulate choices in order to implement sorting. On the other hand, reversing the list is relatively simple since reversing each segment is sufficient to reverse each variant list.

4 Sharing

Sharing in the context of variational lists is valuable, in particular, since it helps to avoid unnecessarily duplicating expensive computations on common list elements. Variational lists provide relatively little value if they simply wrap a set of plain lists. Sharing is one major opportunity in which variational lists can offer an optimization above an ad hoc or brute force approach.

In the representations discussed here, the sharing offered can be classified into one of four categories. Roughly in increasing order of power, these are (1) no sharing, (2) prefix sharing, (3) join sharing, and (4) total sharing. These are explained in turn, each with specific details of the representations that are categorized as such.

No Sharing Of the list representations described here, only the box list offers no sharing at all. It stores each variant completely independently and thus duplicates not only storage costs for each element that could, in theory, be shared, but also offers no mechanism to share computations on common elements.

Prefix Sharing Offering an incremental improvement over no sharing at all, prefix sharing enables sharing common elements only in list prefixes, that is, from the first element up until the first point of variation between two variants. This kind of sharing occurs in the suffix list representation discussed in Section 3.5.

Consider the example suffix list shown in Figure 2e. There `printf("Hello");` is shared, but `printf("Goodbye.");` is not. Prefix sharing is somewhat unpredictable since it depends on the shape of the data being stored.

Table 2. A summary of the type of sharing offered by each representation discussed in this work.

	Sharing Type			
Representation	None	Prefix	Join	Complete
Box	×			
Pointer				×
Choice			×	
Tag			×	
Suffix		×		
Segment			×	

Join Sharing Join sharing is defined by the ability to share common list elements by aligning the separate variants so that common elements occur in the same position. This is the approach taken in choice, segment, and tag lists. Join sharing is sometimes more effective than prefix sharing and never worse.

Consider the example shown in Figure 2c. There, we inserted empty `Nothing` elements, rendered as ϵ in order to allow the variants to share the last token `printf("Goodbye.");`. Padding the lists with empty elements in this manner can increase the amount of sharing substantially.

Unfortunately, finding an alignment that maximizes sharing is difficult and the subject of much research in its own right, particularly in molecular biology [6]. Furthermore, a maximal alignment can be inefficient if too many empty values are padded.

Complete Sharing Pointer lists are the only variational lists we have presented which can share every common element regardless of the number of variants or the positions of particular values. Each element is stored exactly once. Although optimal in a sense, there is inherent overhead.

A summary of the kinds of sharing exhibited by our representations is shown in Table 2.

5 Variational Programming

We can now focus on designing a variational list programming interface. The first step is to define a type class that contains basic list library function analogues. Type classes, as we use them, are essentially like interfaces in Java.

```
class VList vl where
  vEmpty :: vl a
  vCons :: V (Maybe a) -> vl a -> vl a
  vTail :: vl a -> vl a
  ...
```

If we want to implement functions generically for all of our lists, we need some additional variational programming machinery. For example, in order to work with variational `Bool` values (such as those returned from `vNull`) we need a variational `if` construct. We use `VChc` here instead of the generic `V` because it helps to clarify a point of difficulty after the following initial attempt.

```
vIf :: VChc Bool -> VChc a -> VChc a -> VChc a
vIf A⟨l, r⟩ t f = A⟨vIf l t f, vIf r t f⟩
vIf b t f = if b then t else f
```

Suppose we now want to define a generic variational list reversal function. This will take the following (incomplete) form.

```
vRev :: (VList vl) => vl a -> vl a
vRev xs = vIf (vNull xs) vEmpty ...
```

Unfortunately, this definition already contains a type error. We want to pass `vEmpty` to `vIf`, which is a variational list, but that function only accepts choices with the type `VChc`.

What we need instead is to come up with a type that describes variational values more generally, including both applications of our variational type constructor as well as any kind of variational list. This, too, can be achieved with a new type class. Unlike before, this code is unabridged because the details are necessary. `Name`, now revealed, is just a string encoding the dimension of variation.

```
class VVal (v :: * -> *) where
  mkV :: Name -> v a -> v a -> v a
  liftV :: a -> v a
```

Now we need to make both our choices and all kinds of variational lists instances of this new type class. This is trivial and omitted for choices (`VChc`) but the variational list instance is trickier. For `mkV`, we need to take two variational lists and zip them together.

```
instance (VList vl) => VVal vl where
  mkV = mkVL
  liftV = vSingleton . liftV

mkVL :: (VList vl) => Name -> vl a -> vl a -> vl a
mkVL n l r =
  if vAllNull l && vAllNull r
  then vEmpty
  else vCons (liftV n (vHead l) (vHead r))
             (mkVL n (vTail l) (vTail r))
```

Now we can return to our `vIf` definition and modify it to work with values of this type instead of just `VChc`.

```
vIf :: VVal v => VChc Bool -> v a -> v a -> v a
vIf A⟨l, r⟩ t f = mkV A (vIf l t f) (vIf r t f)
vIf b t f = if b then t else f
```

Now, finally, we can implement a variational list reversal function.

```
vRev :: (VList vl) => vl a -> vl a
vRev vl = vIf (vNull vl)
          vEmpty
          (vCat (vrev (vTail vl))
                (vSingleton (vHead vl)))
```

This generic interface, in addition to being useful, enables meaningful benchmarking, which is discussed in the next Section.

6 Performance

The runtime complexity of an algorithm is expressed as a function that depends on the size of its inputs. In the case of variational data structures, runtime not only depends on the

total size of the input, but also on the variability (expressed, for example, as the number of variants) and exploitation of redundancy in the data (expressed, for example, by the degree of sharing).

As an example, consider finding the last element in a plain linked list. Traditionally this takes $O(n)$ time. However, the variational analogue has to deal with potentially many variants of different lengths.

We opt to use $\bar{n} = N/v$ to denote the average length (or size, generally) of a variant list and then write $O(v\bar{n})$ to express that the runtime of an operation is linear with respect to the number of variants and their respective lengths. This is more succinct than using summation notation. Generally, if a function `f :: [a] -> [b]` has complexity $O(g(n))$ where n is the length of the input list, then the lifted version `vF :: V [a] -> V [b]` has time complexity $O(vg(\bar{n}))$.

Since the complexity can depend on the amount of sharing in a variational list, we use the variable S to indicate a metric depending on the redundancy in the data and the corresponding degree of sharing achieved by the data structure. We define $S = 0$ to mean no sharing (but arbitrary data redundancy) and $S = 1$ to mean completely redundant data and total sharing, i.e. every element is stored once. This metric accounts for actual sharing, rather than potential sharing.

For example, suppose we have two variant lists `[1,2,4]` and `[3,2,1]`. Under prefix sharing the ratio is $S = 0$. For a list representation with join sharing, we have $S = 1/3$ since only the `2` will be shared. With a pointer list, we achieve $S = 2/3$ since `1` and `2` can be shared.

6.1 Specific List Operations

Unsurprisingly, sharing can lead to improved performance. What is surprising, however, is that some representations are actually slower than a brute force approach. We have seen this in the `head` function, which requires some representations to potentially traverse the entire length of the list. This effectively turns a constant time operation into one linear with respect to the length of the list. This kind of asymptotic slowdown can occur (for some representations) whenever the operation relies on the assumption that one variational element is equivalent to one plain element, i.e., that it contains one plain value for all variants. Other typical examples include `intersperse` and `zip`.

Conversely, consider `reverse`. Both box lists and pointer lists have worst-case complexity $O(v\bar{n})$ and no improvement in the best case. This is not surprising for box lists since we have no sharing, but the sharing offered by pointer lists does not provide any improvement because we still need to reverse all the copies of pointers.

For the other four representations, however, we need to factor in the amount of sharing that occurs as part of the measure. Using the measure S this gives us a runtime complexity $O(\bar{n} + \bar{n}(v - Sv)) = O(\bar{n}(1 + v(1 - S)))$ for those representations. Plugging in values for S between 0 and 1 gives back the expected results. Although this is constant factor whenever $v \ll \bar{n}$ and easy to dismiss theoretically, the impact on pragmatic performance might still matter.

6.2 Benchmarks

While the computational complexity measures of operations are interesting, they can sometimes fail to fully portray the realities of practical performance. This is the case with our results.

Our benchmarking was performed on files from popular C/C++ projects for which the source is freely available, including Python, GCC, Gecko/Firefox, Linux, and Stockfish.[2] We selected files in each of those source code bases which contained a near-median number of CPP conditional compilation macros. Although we believe these files provide a good overview, it is not possible to choose files that precisely represent the general case since the use of the preprocessor varies tremendously based on the needs and coding style of a project [9].

In files with no or minimal CPP annotations, we would expect performance comparable to the base case for all representations, which is not insightful. On the other hand, files with the highest number of annotations are atypical and sometimes even too slow to benchmark reasonably.

The benchmarks reported here are based on a list filtering operation, a generally useful list operation that requires traversing the entire list. A filtering operation represents a realistic operation to be performed on a source code file, such as finding all fully configured sections in a file which satisfy some predicate (e.g., all parts of a file that apply a particular function or redefine a particular variable).

```
vFilter :: (VList v) => (a -> Bool) -> v a -> v a
vFilter p l = vif (vNull l) vEmpty (fil p l)
  where fil p l = vCons (vFilterV p (vHead l))
                        (vFilter p (vTail l))

vFilterV :: (a -> Bool) -> V (Maybe a) -> V (Maybe a)
```

We used the Criterion[3] package to benchmark our Haskell-based implementation of these list representations on 2.8Ghz Intel Core i5 with 16GB of memory. We use the default settings which run each benchmark 1000 times in order to ensure a dependable result in the presence of factors like garbage collection. Table 3 shows the benchmarked result for applying `vFilter` with a predicate for including an arbitrary line of code to one file from each of the five repositories. It also includes information about the number of lines of code (*LoC*) and lines of relevant CPP annotations (*LoCPP*) for each of the files.

The runtimes show some unexpected results. First, we can see that not every representation is able to outperform the box list base case. Most notably, both pointer lists and

[2] www.python.org; gcc.gnu.org; www.mozilla.org; www.kernel.org; https://stockfishchess.org

[3] hackage.haskell.org/package/criterion

Table 3. Benchmark results for applying `vFilter` to five CPP-annotated source code files for each of our list representations. Runtimes given in μs.

List Representation	`parser.c` (Python)	`bitmap.c` (GCC)	`FilterProcessing.cpp` (Firefox)	`raw.c` (Linux)	`bitboard.h` (Stockfish)
LoC/LoCPP	488/22	2280/16	262/26	1369/21	334/9
Box	1220	894	336	3610	1680
Pointer	341000	12800	321	3490	55600
Choice	1770	486	436	909	357
Suffix	1090	465	363	823	620
Segment	22	462	41	377	152
Tag	11000	438	2870	11300	6830

tag lists occasionally perform significantly worse than the base case. Conversely, choice and suffix lists frequently (but not always) offer a noteworthy improvement over the base case. However, from the results, it is clear that the segment list representation is the top performer, beaten only in one benchmark and even then only by a small margin.

As mentioned, both choice and segment lists suffer from the potential for asymptotic slowdown in functions like `vHead`. The benchmarks here seem to suggest that this does not manifest in practice in at least some circumstances and that we should probably rely more on benchmarks than on theoretical performance.

6.3 Interface Design

One further optimization can come from a refactoring of our implementation. By making our type classes more granular, we can provide more efficient implementations for many of the list operations. For example, instead of a filter function based on the potentially slow `vHead` and `vTail` functions, we can define the following.

```
class VFilter vl where
  vFilter :: (a -> Bool) -> vl a -> vl a
```

Then we can instantiate this type class for all of the list representations.

We re-benchmarked this more granular use of type classes on the most inconsistent representations, the results of which are shown in Table 4.

Performance is now generally improved and more consistent. Not all the news is good, however. By comparing the two tables, we can see that in three of the five cases, the type class design actually worsens the performance of the segment lists. In the remaining two cases performance is more or less unchanged.

7 Related Work

There is relatively little work so far exploring variational data structures. Erwig and Walkingshaw [11] developed a general strategy of adding variation to data types and also applied to to lists. That work directly informs the suffix lists described here. Walkingshaw et al. [24] explored some of the possibilities of variational lists, sets, and maps. While that work broadly covered many data structures and possibilities, we focus more deeply on lists and practical list programming. Meng et al. [21] designed and evaluated the performance of variational stacks. Erwig et al. [12] investigated variational graphs using a tag-based approach much like the tagged list representation discussed here.

Despite the relatively small amount of work specifically targeting variational data structures, there is a lot of work within the larger fields of variation research that shares some common goals. Both TypeChef [16] and SuperC [13] make use of variational data structures to perform variability-aware code analysis. In the same way that variational data structures are concerned with sharing as much as possible between otherwise independent variants, variational model checking work also looks to avoid unnecessary duplication of analysis [2, 8, 23]. Some model checking work has even gone so far as to effectively design variational data structures, even if not describe as such directly [7, 18].

Although not principally focused on variational data structures, Liebig et al. [20] suggest keeping variability as local as possible and specifically consider the differences between a choice of map structures and a map of variational values. Similar ideas regarding localizing variation and the notions *late splitting* and *early joining* are expressed by both Kästner et al. [14] and Apel et al. [1].

Finally, some work on testing variational code has tried to avoid a brute force approach by variationalizing an interpreter [15, 17, 22]. Similar approaches have also been proposed for dealing with privacy policies [3, 4, 25].

8 Conclusions

Our exploration of variational list data structures has two main take-away messages. First, we have learned that the segment list representation (paired with the generic variational list interface) outperforms our other representations. This suggests that the best approach for a practical variational list library would be one based only on segment lists with no real need for the notion of a generic `VList` that can be instantiated to several different concrete types. As a corollary,

Table 4. Benchmarks for applying `vFilter` on some list representations using a more granular approach. All runtimes given in μs. Compare to Table 3.

List Representations	`parser.c` (Python)	`bitmap.c` (GCC)	`FilterProcessing.cpp` (Firefox)	`raw.c` (Linux)	`bitboard.h` (Stockfish)
Pointer	1190	1000	420	3980	1940
Choice	176	456	169	473	158
Segment	173	451	168	473	162
Tag	172	490	166	470	161

we recommend that designers of other variational data structures perform pragmatic benchmarking, because theoretical worst-case analysis may not align with the results.

Second, we have learned a couple of lessons about designing variational data structures in general. Through the general failure of the pointer lists, we have learned that hash-consing (at least in pure languages) is likely not worth the extra overhead. Hybrid approaches such as storing the indices in a segment list offer no improvement.

We have also seen that even though our segment lists eventually outperformed everything else, a library design that avoids over-generalizing is preferable in most cases. For example, if we were to move on to implementing variational trees or graphs, and were unable to find a representation that outperforms the others, we would expect the best approach to be supporting multiple representations and having one type class for every operation that is supported.

References

[1] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Grösslinger, and Dirk Beyer. 2013. Strategies for Product-line Verification: Case Studies and Experiments. In *IEEE Int. Conf. on Soft. Eng.* 482–491.

[2] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions using Feature-Aware Verification. In *Int. Conf. Automated Soft. Eng.* 372–375.

[3] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. 165–178.

[4] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proc. ACM SIGPLAN Work. on Programming Languages and Analysis for Security*. ACM, 15–26.

[5] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. Proactive Detection of Collaboration Conflicts. In *Europ. Software Engineering Conf./Foundations of Software Engineering*. 168–178.

[6] Humberto Carrillo and David Lipman. 1988. The Multiple Sequence Alignment Problem in Biology. *SIAM J. Appl. Math.* 48, 5 (1988), 1073–1082.

[7] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. 2011. Symbolic Model Checking of Software Product Lines. In *ACM Int. Conf. on Software Engineering*. ACM, 321–330.

[8] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. 2008. Delta Execution for Efficient State-Space Exploration of Object-Oriented Programs. *IEEE Trans. Softw. Eng.* 34, 5 (2008), 597–613.

[9] Michael D. Ernst, Greg J. Badros, and David Notkin. 2002. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. on Software Engineering* 28, 12 (Dec 2002), 1146–1170.

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

[11] M. Erwig and E. Walkingshaw. 2013. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering (LNCS 7680)*. 55–100.

[12] M. Erwig, E. Walkingshaw, and S. Chen. 2013. An Abstract Representation of Variational Graphs. In *ACM Int. Workshop on Feature-Oriented Software Development*. 29–40.

[13] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*. 323–334.

[14] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-aware Testing. In *ACM Int. Work. on Feature-Oriented Soft. Dev.* 1–8.

[15] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. 2012. Toward Variability-Aware Testing. In *GPCE Workshop on Feature-Oriented Software Development*. 1–8.

[16] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: Toward Type Checking # ifdef Variability in C. In *Int. Work. on Feature-Oriented Soft. Dev.* ACM, 25–32.

[17] C. H. P. Kim, S. Khurshid, and D. Batory. 2012. Shared Execution for Efficiently Testing Product Lines. In *Int. Symp. Software Reliability Engineering*. 221–230.

[18] Kim Lauenroth, Klaus Pohl, and Simon Töhning. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *IEEE/ACM Int. Conf. on Automated Software Engineering*. 269–280.

[19] Yun Young Lee, Darko Marinov, and Ralph E. Johnson. 2015. Tempura: Temporal Dimension for IDEs. In *ACM Int. Conf. Soft. Eng.*

[20] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *ACM Symp. on the Foundations of Soft. Eng.* 81–91.

[21] Meng Meng, Jens Meinicke, Chu-Pan Wong, Eric Walkingshaw, and Christian Kästner. 2017. A Choice of Variational Stacks: Exploring Variational Data Structures. In *ACM Int. Workshop on Variability Modelling of Software-intensive Systems*. 28–35.

[22] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *ACM Int. Conf. Soft. Eng.* 907–918.

[23] Hendrik Post and Carsten Sinz. 2008. Configuration Lifting: Verification meets Software Configuration. In *Int. Conf. Automated Soft. Eng.* 347–350.

[24] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. 2014. Variational Data Structures: Exploring Tradeoffs in Computing with Variability. In *ACM Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*. 213–226.

[25] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. 85–96.

[26] YoungSeok Yoon and Brad Myers. 2015. Supporting Selective Undo in a Code Editor. In *ACM Int. Conf. Soft. Eng.* 223–233.