Techniques in Variability-aware Data Structures Supervised by *R. Shahin* and *M. Chechik*

Andrew Blinn
University of Toronto
`me@andrewblinn.com`

April 5, 2019

Abstract

In 2014 Walkingshaw et al [1] put out the call for exploring variability in software at the data structures level. To this effect, Ramy Shahin implemented a Haskell-based framework intended to generically lift extant data structures to their variational equivalents, focused in particular on variability-aware testing and verification of software product lines. This semester, I worked with Ramy on extending and testing this implementation. My contributions include (1) a tool for visualizing sharing in data structures using Graphviz, (2) theoretical work on 'deep lifting' existing functions to efficient variational versions, aided by Racket/Redex, and (3) a tool leveraging the TypeChef variational C preprocessor to marshal existing variational source code as real-world test-cases.

***K*eywords** Data Structures · Variational Data Structures · SPL · Haskell

1 Variational Data Structures and Software Product Lines

To concretely explain the need for variational data structures, imagine a C-based software product line with 100 IFDEF preprocessor directives, subtending a space of 2^{100} variations. Any naive exhaustive static or dynamic analysis of this space is not feasible. The only way forward is to exploit the shared structure between variations. In practice, such variational analyses tend to be ad hoc.

In an ideal world, we could take an existing token counter or data flow analyzer designed to operate on a single source file or AST and automatically 'lift' it into a version designed to efficiently process multiple variants of a single source, with the possible gains being bounded by the amount of commonality between variations.

Functional languages in general, and Haskell in particular, provides an interesting environment to prototype these ideas. First, as a pure functional language, native Haskell data structures are persistent; in particular, the Haskell compiler GHC does not unnecessarily duplicate data when new structures are composed from already-instantiated substructures; see Section 3 for some concrete examples. Second, Haskell evaluation semantics are lazy; computations are only performed if there is a data dependency path linking them to the final result. Third, Haskell typeclasses present a natural, lightweight interface for extending existing data-types.

Ramy's existing work in Haskell consists primarily of the `Var` type. This abstract type takes an existing type as a parameter; for example `Var Int` represents the concrete type of variational integers. Var is implemented as an instance of the Applicative typeclass, which means that it does its primary work through the `apply`/<*> method, which is essentially an overloaded function application whereby a variational function is applied to variational data. There is also an additional method, `mkVarT`/`pure`, which wraps a single variant with the presence condition `True`.

On a basic level, a `Var` instance is a map/dictionary from presence conditions to particular variants. Presence conditions are propositional formulas whose atoms are atomic propositions called features. These features could be, for example, the IFDEF flags in a C/C++ based SPL. One common issue in variational processing is the creation of irrelevant intermediate results; that is, variants whose presence conditions are unsatisfiable. To prevent doing unnecessary work in unsatisfiable branches, Ramy's `apply` method uses binary decision diagrams via CUDD [9] to prune unsatisfiable variants as they are created.

Another of Var's apply-level optimizations is compaction, which introspects on the memory location of variants and make sure that identical variants and their presence conditions are merged, preventing redundant computation.

VList is a specific instance of Var, lifting Haskell's native linked-list type to a variational version. The list constructor cons is lifted using apply to a variational version, vCons, enabling the optimizations above, and (with compaction enabled) having the effect of collapsing shared list suffixes into a single copy.

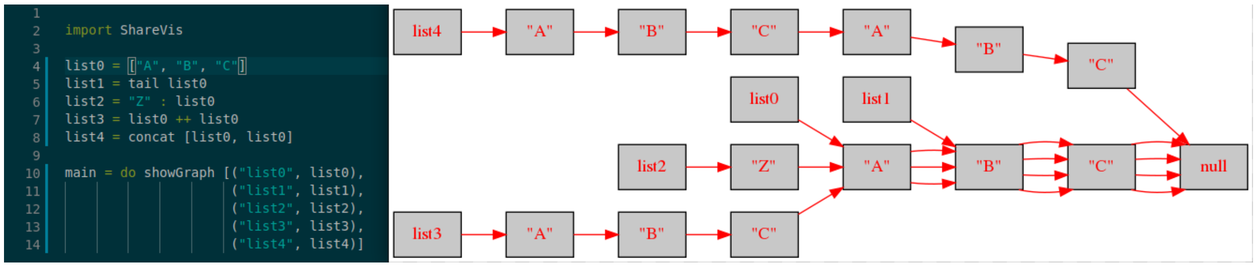
2 My contributions

My concrete contributions include:

- Implementing ShareVis: This is a tool which uses Graphviz [2] to visualize substructural sharing in Haskell data structures; in particular, this was used to verify data sharing in Ramy's Var typeclass after compaction. This is a small but functionally complete standalone tool, which I've made available through my github since it might have external use as a didactic tool: github.com/disconcision/spyshare
- Developing an abstract system of rewriting rules to deep lift functions to efficient variational versions, modelling these rules in Racket/Redex, and beginning an implementation of these rules in Haskell using GHC Rewrite Rules. The principle here is using source-level rewriting to avoid redundant computation. This is incomplete, owing mostly to the complexity of iterating on candidate approaches based on complex feedback from the compiler, but I believe the approach is promising.
- Developing a preprocessor wrapper leveraging TypeChef to convert C source with IFDEF pragmas into Ramy's Var typeclass.

3 Visualizing data sharing with ShareVis

ShareVis uses the StableNames [3] package to assign unique identifiers for the memory locations used by a Haskell program. I designed a simple Haskell typeclass, MemMappable, which abstracts the process of using StableName to create a graph representing the memory locations used by an instance of a particular algebraic data type. I use Graphviz to generate an image of the graph, and ImageMagick to display as part of an integrated process. By providing an instantiation of MemMappable for a particular data structure, and end-user can visualize substructural sharing in a data structure of their choice.

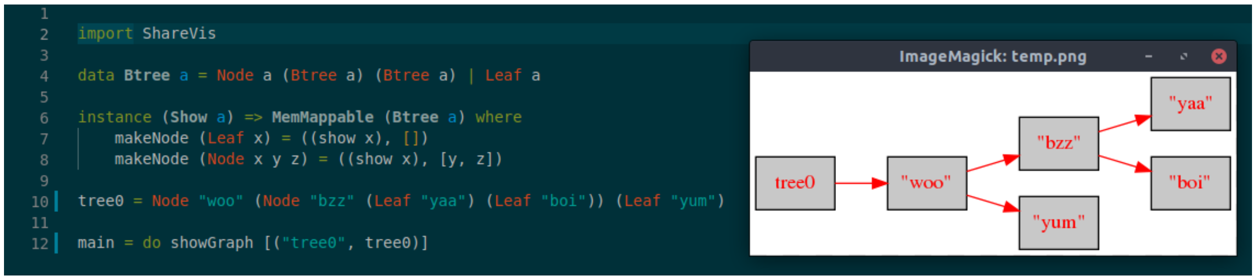


```haskell
import ShareVis

list0 = ["A", "B", "C"]
list1 = tail list0
list2 = "Z" : list0
list3 = list0 ++ list0
list4 = concat [list0, list0]

main = do showGraph [("list0", list0),
                     ("list1", list1),
                     ("list2", list2),
                     ("list3", list3),
                     ("list4", list4)]
```

Figure 1: ShareVis for native lists. Note how defining using Haskell's append method (++) results in sharing, whereas the more abstractly implemented concat method results in duplication



```haskell
import ShareVis

data Btree a = Node a (Btree a) (Btree a) | Leaf a

instance (Show a) => MemMappable (Btree a) where
    makeNode (Leaf x) = ((show x), [])
    makeNode (Node x y z) = ((show x), [y, z])

tree0 = Node "woo" (Node "bzz" (Leaf "yaa") (Leaf "boi")) (Leaf "yum")

main = do showGraph [("tree0", tree0)]
```

Figure 2: ShareVis for a custom binary tree ADT, including the MemMappable instance for the ADT

Going forward, ShareVis provided a quick way to verify that sharing due to pruning and compaction were in fact occurring.

4 Shallow versus deep lifting: An introduction

One simple desideratum for variational programs is that we should be able to avoid the combinatorial[1] explosion of our function call hierarchy where not demanded by the runtime data. Consider the following function definition:

```
bang :: Int -> Int -> Int
bang a b = (+) (foo a) (bar b)
```

Recall that our parametric `Var` type is implemented as an instance of the `Applicative` typeclass, permitting the automated lifting of values of type `a` to `Var`, functions of type `a -> a` to `Var a -> Var a`, etc. as follows:

```
shallowBang :: Var Int -> Var Int -> Var Int
shallowBang = liftA2 bang
```

This lifting is 'shallow' in the sense that the variability exists entirely on the outside of the function's implementation. Consider a call to `shallowBang a b` given two variational values `a` and `b`, where `a` has 5 distinct variations and `b` has only one. Despite the fact that `b` has no actual variability, the entire function body will be executed `|a|*|b| = 5` times, and hence 5 calls to `bar` will be made when only one is conceptually required. Now consider the following deeply-lifted variation:

```
deepBang :: Var Int -> Var Int -> Var Int
deepBang va vb =  (liftA2 (+)) (liftA foo va) (liftA bar vb)
```

Here, we have moved the locus of variability into the function itself. Given the previously described inputs, `bar` will be called only once on the unique variant of `b`. Notice that we have accomplished this manual deep-lifting by performing a shallow lifting of each call made in `bang`. A 'full' deep-lifting would carry this process down into the individual calls, recursively deep-lifting them, down to the point of certain primitive operations which may have custom lifted implementations.

To establish a baseline and prepare for more detailed testing, we used Haskell's profiling functionality [6] to verify the above assertions. Specifically, the RTS flag tells the runtime system to trace the callstack and track the number of calls to each function, confirming that the following calls to `shallowBang` and `deepBang` involve 4 and 1 calls to `foo` respectively. Below, `p` and `q` are features; `pq` represents their conjunction, `_p_q` represents the conjunction of their negations, etc. Together they represent 4 mutually-exclusive presence conditions conjoining to $true$, which itself is represented by the presence condition `tt` on the next line:

```
a = mkVars [(1, pq), (2, p_q), (3, _pq), (4, _p_q)]
b = mkVars [(20, tt)]
shallowBang a b
deepBang a b
```

The above implementation of `deepBang` constitutes a manual deep lifting. While not quite as straightforward as a shallow lifting, the process is still fairly mechanical and should be susceptible to automation. We investigated approaches based on memoization, in-lining, rewrite rules, and full syntax-rewriting via Template Haskell.

5 Deep Lifting in Haskell: Two Approaches

I began with some basic experiments in Template Haskell, a template-based metaprogramming tool for GHC [4]. Template Haskell is powerful, but notoriously complex. It is also not designed directly for syntax-rewriting, being more oriented toward code generation. I determined that it could likely accomplish the rewriting we needed, at least in important special cases, but that this rewriting would involved laboriously marking up target code. This job which would need to be handled by a separate tool, and might necessitate the handling the entirety of Haskell's complex surface syntax. There is likely existing work which can be leveraged to streamline these issues, but we decided to consider another option.

[1]nb: My spellcheck does not recognize this, but suggests 'gubernatorial'

To that end, I explored a mechanism to achieve deep-lifting based on GHC's rewrite rules. Rewrite rules were introduced by Peyton Jones, Tolmach, and Hoare [5] as a library-level mechanism for end-users to augment the compiler. The titular rewrite rules are user-provided equations written in Haskell's surface syntax, directing GHC to replace the expression on the LHS with a more efficient variation on the RHS. Here is an example from the documentation [7], where mapping twice over a list with two functions is replaced by mapping a single time with their composition:

```
{-# RULES
  "map/map"  forall f g xs.  map f (map g xs) = map (f.g) xs
#-}
```

Rewrite-rules offer the ability to interleave with other compiler optimizations in a controlled way via their phase system, which benefits us in two ways. First, certain kinds of in-lining can be either forced or suspended until before/after certain rules are applied. Second, rewrite rules make it possible - in principle - to conduct rewriting after Haskell code has been reduced to Core, GHC's simplified internal language - essentially an augmented lambda calculus. This is would allow us to avoid having to special-case on Haskell's complex surface syntax, working around complexities like guards and pattern matching.

However, they also have limitations, mostly owing to their simplicity. The top level of each rule must be a function application. This means directly rewriting function definitions is a no-go; rewriting must take place at the call site. This means that coordinating with the in-liner is obligatory, in order to expose the bodies of the to-be-lifted functions to the rewriter. Additionally, rewriting is nondeterministic - for cases in which multiple rules match a piece of syntax, the end-user cannot choose the order in which rules will be applied. It is not even determined (at the specification level) whether rewriting proceeds in a top-down or bottom-up fashion. So, for this method to be robust, we would need a set of rewriting rules which are provably confluent.

6 Deep Lifting: Candidate Rewrite Rules

For simplicity I decided on a restricted version of deep lifting based on the `deep`/`shallowBang` example given in the previous section. First, I would concern myself with ecumenical shallow-lifting via the `mkVar` constructor; that is, shallow-lifting functions with trivial presence conditions. Second, I restricted the object-level language to be lifted to following restricted subset of Haskell:

```
e::= (e e) -- unary function applications
     (\ id -> e) -- unary lambdas
     (\ id id -> e) -- binary lambdas
     x -- identifier references
     number -- numeric literals
     (apply e e) -- apply method of Var
     (mkVarT e) -- mkVarT method of Var
```

My general approach in designing rewriting rules was to identify shallowly lifted applications in the source, inline the function term in the form of a lambda, and then 'lift' each regular function application out of the lambda, where it would become a call to the `apply` method:

```
bang a b = (+) (foo a) (bar b)

-- 0. shallowly lifted application
(apply (apply (mkVarT bang) va) vb)

-- 1. inline bang
(apply (apply (mkVarT (\ a b -> ((+ (foo a)) (bar b)))) va) vb)

-- N. desired deep lifting
(apply (apply (mkVarT +) (apply (mkVarT foo) va)) (apply (mkVarT bar) vb))
```

Note that the final expression `N` above is the same as the deep-lifted bang from the previous section, except that the more abstract `liftA` and `liftA2` methods have been replaced by the primitive `apply` and `mkVarT` methods with which it is implemented. I used the former methods earlier to reduce syntactic baggage, but I will consistently use the latter from here out so as to minimize the number of rules needed.

The central rule I developped is the following, expressed as a Haskell rewrite rule:

```
{-# RULES
"r3" forall vx vy A B . apply (apply (mkVarT (\x y -> (A B))) vx) vy
                      = apply (apply (apply (mkVarT (\x y -> A)) vx) vy)
                              (apply (apply (mkVarT (\x y -> B)) vx) vy)
#-}
```

The outermost apply on the rule's LHS is actually the lifted form of the application (A B) inside the original lambda. Note how the lambda parameters, and the variation data vx to which they are applied, are 'pushed down' onto A and B, the subterms of the original application. In the case where A or B is itself an application, the above rule can be applied recursively. In the case where A or B is simply a literal or variable reference, the created applications are superfluous and can be dispensed with via the following two rules:

```
{-# RULES
"r1a" forall vx. apply (apply (mkVarT (\x y -> x)) vx) vy = vx
"rba" forall vx. apply (apply (mkVarT (\x y -> y)) vx) vy = vy
"r2" forall vx _A . apply (apply (mkVarT (\x y -> A)) vx) vy = (mkVarT A)
#-}
```

Interestingly, these latter rules are respectively identical to or follow immediately from the first two Applicative laws [11], rules which Haskell mandates (but cannot automatically verify) that Applicative instances should follow. This, coupled with the conceptual simplicity of the central rule, leads me to suspect that there is some more abstract way of characterizing the process of deep lifting, but I was not immediately able to tie this to existing work.

7 Modelling the Rewrite Rules in PLT Redex

Note that rule r3 above leads to an at-least-temporary blowup in expression size. In order to manage this complexity and avoid error, I employed Redex [8], a Racket-based tool for semantics engineering. First, I specified the syntax of the restricted language from the previous section in Redex:

```
(define-language L
  (e (e e)
     (\ x -> e)
     (\ x x -> e)
     x
     number
     (apply e e)
     (mkVarT e))
  (t (→ t t)
     (Var t)
     Int)
  (x variable-not-otherwise-mentioned))
```

Then, I specified evaluations contexts. Redex evaluations contexts provide a declarative language for specifying the recursion scheme Redex will use to apply reduction rules. The following basically specify that reduction rules can be applied to any constituent subexpressions in any order. The second top-level subexpression (beginning with v) indicates the values of the language, providing termination conditions for the recursion.

```
(define-extended-language Ev L
  (E (e E)
     (E e)
     (mkVarT E)
     (apply e E)
     (apply E e)
     hole)
  (v (\ x -> e)
     (\ x x -> e)
     number))
```

After specifying the language syntax and evaluations contexts, I was then free to define reduction rules and let Redex visualize the non-deterministic application of those rules to provided expressions. For space purposes, I'll reproduce only the central rule here. The rest are available at github.com/disconcision/vardatalab/rewrite.rkt

```
(define red
  (reduction-relation
   Ev #:domain e
   (--> (in-hole E (apply (apply (mkVarT ( (x_1 t_1) (x_2 t_2) -> (e_1 e_2))) e_11) e_22))
        (in-hole E (apply (apply (apply (mkVarT ( (x_1 t_1) (x_2 t_2) -> e_1)) e_11) e_22)
                          (apply (apply (mkVarT ( (x_1 t_1) (x_2 t_2) -> e_2)) e_11) e_22)))
        "r3")))
```

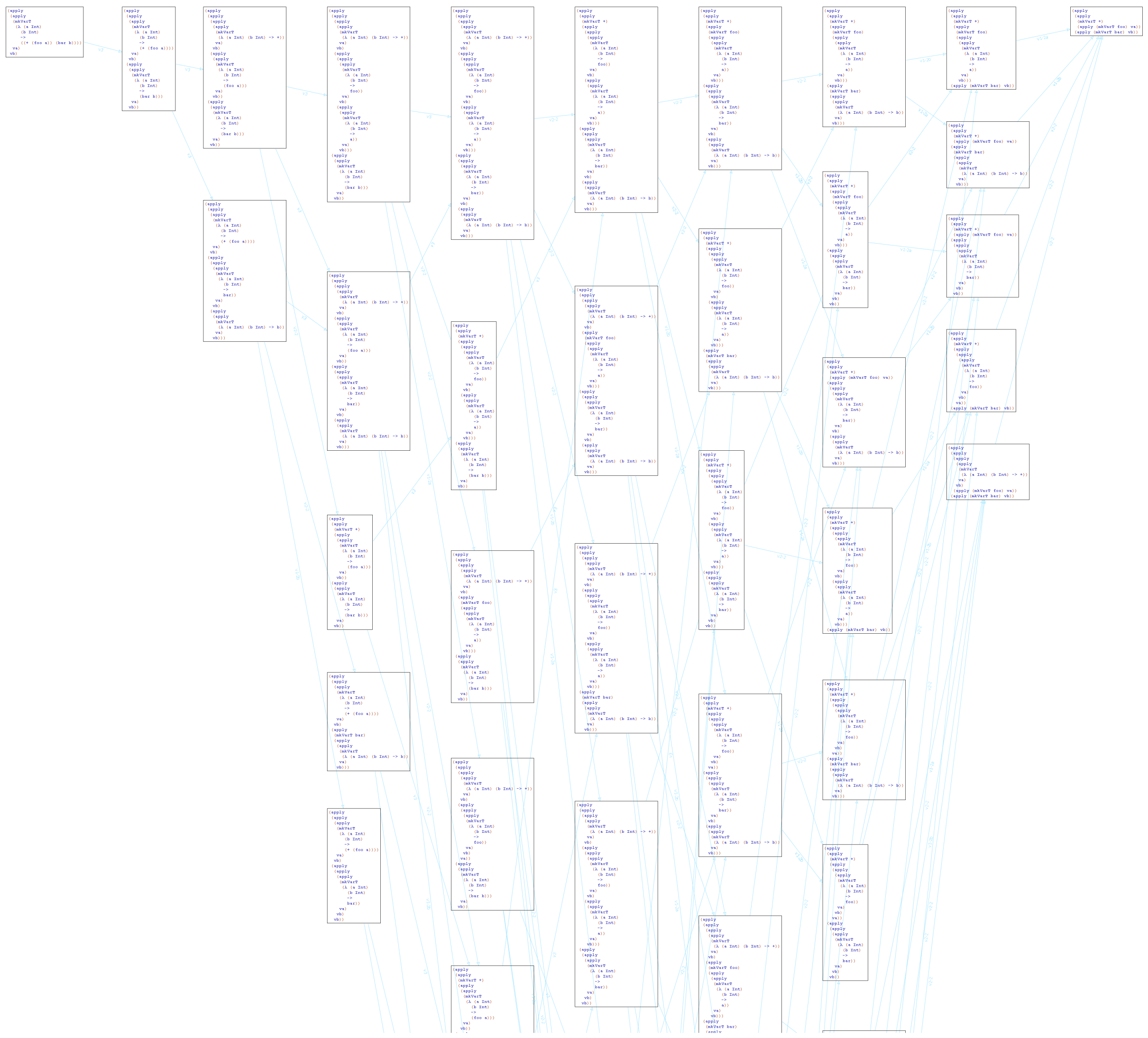
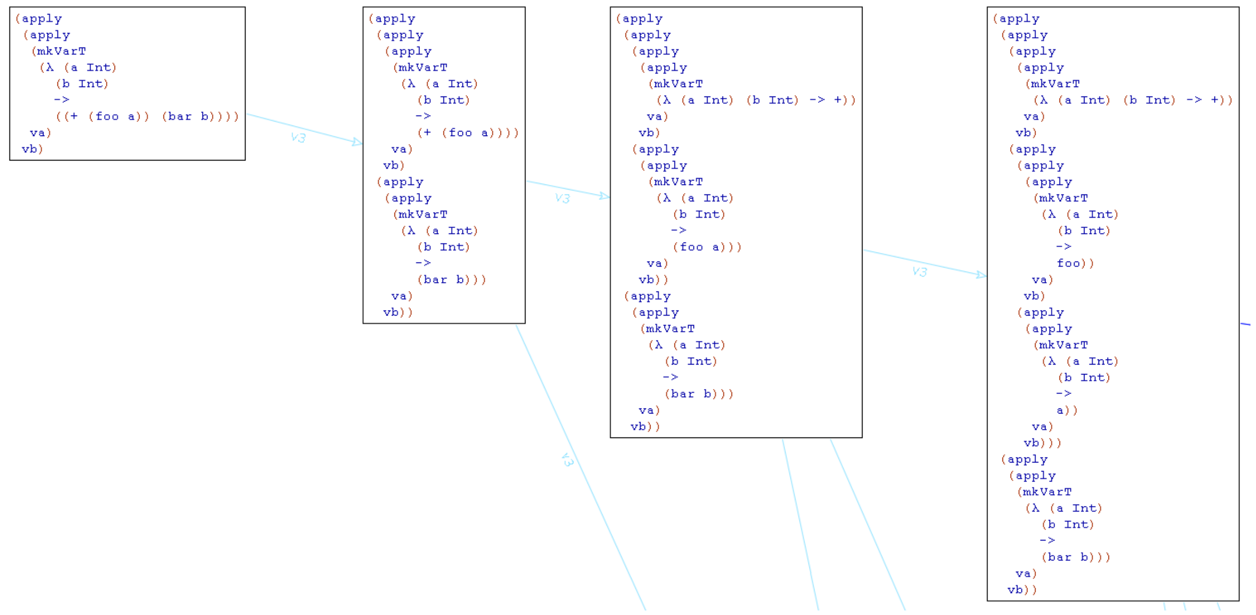


Figure 3: The following figure is a vertically truncated depiction of the entire expansion taking the shallowly lifted bang function from the previous section (upper left) to the deeply lifted version (upper right). More legible details are on the next page. Here we can see Redex step the expression non-deterministically through 56 distinct stages.

Experimenting with Redex allowed me to quickly determine the viability of candidate rewriting rules, and also provided evidence to suggest that the rules are confluent; in the above case, you can see that despite taking multiple paths, all reductions terminate on the desired deep-lifting.



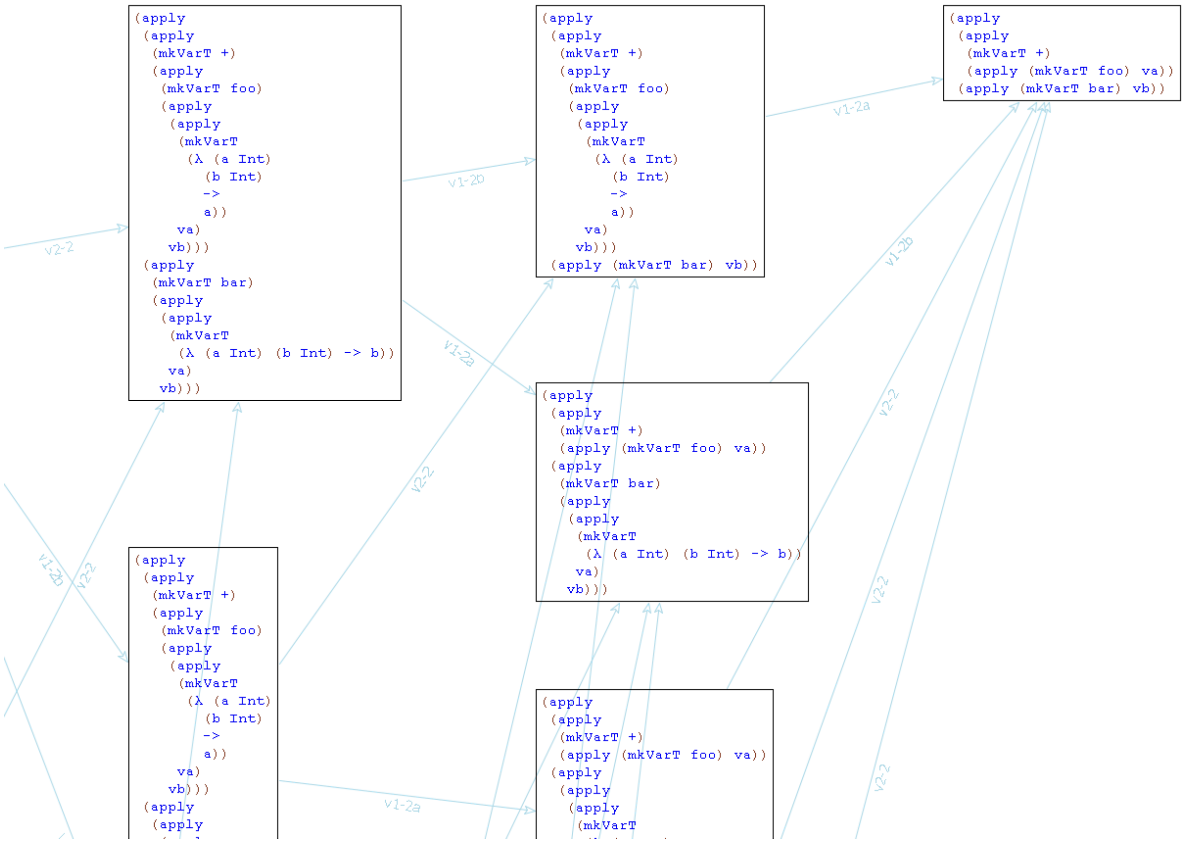
```
(apply
 (apply
  (mkVarT
   (λ (a Int)
     (b Int)
     ->
     ((+ (foo a)) (bar b))))
  va)
 vb)
```

```
(apply
 (apply
  (apply
   (mkVarT
    (λ (a Int)
      (b Int)
      ->
      (+ (foo a))))
   va)
  vb)
 (apply
  (apply
   (mkVarT
    (λ (a Int)
      (b Int)
      ->
      (bar b)))
   va)
  vb))
```

```
(apply
 (apply
  (apply
   (apply
    (mkVarT
     (λ (a Int) (b Int) -> +))
    va)
   vb)
  (apply
   (apply
    (mkVarT
     (λ (a Int)
       (b Int)
       ->
       (foo a)))
    va)
   vb))
 (apply
  (apply
   (mkVarT
    (λ (a Int)
      (b Int)
      ->
      (bar b)))
   va)
  vb))
```

```
(apply
 (apply
  (apply
   (apply
    (mkVarT
     (λ (a Int) (b Int) -> +))
    va)
   vb)
  (apply
   (apply
    (apply
     (mkVarT
      (λ (a Int)
        (b Int)
        ->
        foo))
     va)
    vb)
   (apply
    (apply
     (mkVarT
      (λ (a Int)
        (b Int)
        ->
        a))
     va)
    vb)))
 (apply
  (apply
   (mkVarT
    (λ (a Int)
      (b Int)
      ->
      (bar b)))
   va)
  vb))
```

Figure 4: The first four stages of applying the specified rewrite rules to the inlined, shallowly-lifted `bang` function (upper left). Note that the edges are labelled with the relevant rule.



```
(apply
 (apply
  (mkVarT +)
  (apply
   (mkVarT foo)
   (apply
    (apply
     (mkVarT
      (λ (a Int)
        (b Int)
        ->
        a))
     va)
    vb)))
 (apply
  (mkVarT bar)
  (apply
   (apply
    (mkVarT
     (λ (a Int) (b Int) -> b))
    va)
   vb)))
```

```
(apply
 (apply
  (mkVarT +)
  (apply
   (mkVarT foo)
   (apply
    (apply
     (mkVarT
      (λ (a Int)
        (b Int)
        ->
        a))
     va)
    vb)))
 (apply (mkVarT bar) vb))
```

```
(apply
 (apply
  (mkVarT +)
  (apply (mkVarT foo) va))
 (apply (mkVarT bar) vb))
```

```
(apply
 (apply
  (mkVarT +)
  (apply (mkVarT foo) va))
 (apply
  (mkVarT bar)
  (apply
   (apply
    (mkVarT
     (λ (a Int) (b Int) -> b))
    va)
   vb)))
```

```
(apply
 (apply
  (mkVarT +)
  (apply
   (mkVarT foo)
   (apply
    (apply
     (mkVarT
      (λ (a Int)
        (b Int)
        ->
        a))
     va)
    vb)))
 (apply
  (apply
```

```
(apply
 (apply
  (mkVarT +)
  (apply (mkVarT foo) va))
 (apply
  (apply
   (apply
    (mkVarT
```

Figure 5: Rewriting converging to the deeply-lifted version (upper right)

One complication with these rewriting rules is that there needs to be a version of each of the rules for lambdas of each arity. This is a likely-insurmountable limitation in the expressiveness of the rewriting rules system, but not necessarily a deal-breaker for our purposes. While I'm hopeful that these rewriting rules will do the trick, I've haven't been able to get them to fire in-sync. I'm confident that this is doable, and my issues thus far are due to an incomplete understanding of GHC's phase system, but I haven't been able to develop an efficient workflow for iterating on this. I essentially have to grep through megabyte-sized compiler AST dumps to see why certain rules fire in certain situations. More work is required.

8 Test case marshalling

In order to test the various optimization of `Var`, and (eventually) compare shallow and deep-lifted variants, we needed a way to automatically convert real-world SPLs into variadic data within Haskell.

While our goal is to also lift existing Haskell-based static analyses, our more modest initial goal is to perform some simple hand-rolled analysis on simple hand-rolled SPLs. Specifically, we wanted to take Ccode with IFDEF-based variability, and perform simple variadic analysis on the lexical level, such as token-counting.

To this effect, we leveraged existing work in the form of TypeChef [10], a Java/Scala-based project for examining variability in C code. Our first goal was to coax TypeChef to produce a list of tokens coupled with the features for which those tokens are present. Since TypeChef is explictly a tool for type-checking and safely refactoring IFDEF-variable C code, we knew it had to do some variability-aware preprocessing. I found a flag to enable a dump after pre-processing, but the data was quite dirty and would require substantial and likely brittle parsing to extract the data we required.

To proceed, I forked the codebase and greped my way upstream, starting with the preprocessor dump flag, until I reached the tokenizer data structure, which turns out to be part of an older preprocessor which TypeChef has wrapped. After experimenting with the internal token data type, I modified the tokenizer code to dump the requsite information to Stderr, and made a bash script to pipe that data to a Haskell post-processor.

In that post-processor, I read in the token data. There was still some parsing involved, as TypeChef also expresses its presence conditions as propositional formulas of atomic features, which I had to serialize as text. I used Parsec [12], a Haskell Parser-Combinator package, to parse these presence conditions to a Haskell data type.

Finally, I automated conversion of this post-processed data to our VList variational lists type. My fork of typechef is available at github.com/disconcision/TypeChef, and a binary JAR is included in my fork of Ramy's SPL library here github.com/disconcision/ProductLineAnalysis.

9 Conclusion

Over the last several months, I've developed two concrete tools extending Ramy's work on variational data structure for SPLs. I hope that ShareVis might even prove useful in a more general context; I've already briefly shown it to a few CSC324 students (in my capacity as a TA) by way of illustrating persistent data structures in Haskell. I hope that Ramy will find my preprocessing tool of use in future comparative benchmarking. Setting up this tool was in some ways the most enlightening part of the project for me; most of my education has been about understanding concepts in a top-down way; having to dig through a large codebase and hack away with very fragmentary understanding to achieve a practical goal was eye-opening - simultaneously frustrating and empowering.

Although incomplete, I greatly benefited personally from my work on rewriting rules, coming away from it with a greater understanding of the Haskell compiler and with experience using interesting tools like Redex. I generally feel like I could have accomplished more. If I was starting over, I would've met more at the beginning, asking more (and stupider!) questions to test and expand my knowledge, as I feel I ended up spending too much time struggling with sometimes-subtle misunderstandings. Regardless, Ramy has been an excellent advisor throughout, and I've had a great time helping him with an interesting project touching both software development theory and practice.

References

[1] Variational Data Structures: Exploring Tradeoffs in Computing with Variability
https://dl.acm.org/citation.cfm?id=2661143

[2] Graphviz - Graph Visualization Software
https://www.graphviz.org/

[3] System.Mem.StableName - Hackage - Haskell
hackage.haskell.org/package/base/docs/System-Mem-StableName.html

[4] Tim Sheard, Simon Peyton Jones. Template meta-programming for Haskell
https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/

[5] Simon Peyton Jones, Andrew Tolmach, Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC
https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc

[6] Glasgow Haskell Compiler User's Guide: Profiling
https://downloads.haskell.org/ ghc/latest/docs/html/users$_{g}uide/profiling.html$

[7] Haskell Documentation 7.14: Rewrite rules
https://downloads.haskell.org/ ghc/7.0.1/docs/html/users$_{g}uide/rewrite-rules.html$

[8] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler - Run Your Research: On the Effectiveness of Lightweight Mechanization
http://users.cs.northwestern.edu/ robby/lightweight-metatheory/

[9] CUDD: CU Decision Diagram package
https://github.com/ivmai/cudd

[10] TypeChef: Variational Analysis for C
https://github.com/ckaestne/TypeChef

[11] Haskell: Applicative functors : Applicative laws
https://en.wikibooks.org/wiki/Haskell/Applicative$_{f}unctorsApplicative_{f}unctor_{l}aws$

[12] Parsec: Monadic parser combinators for Haskell
http://hackage.haskell.org/package/parsec