TECHNIQUES IN VARIABILITY-AWARE DATA STRUCTURES

Andrew Blinn
supervised by R. Shahin and M. Chechik

My Contributions

1. **ShareVis**
   Visualizing Substructural Sharing with Graphviz

2. Candidate Rules for **Deep-Lifting** Variational Programs
   Syntax-rewriting using GHC Rewrite Rules and PLT Redex

3. TypeChef-based **Test Harness**
   Converting variational C code to the VList type

ShareVis : Visualizing Sharing

- Immutability enables persistent data structures
- Variability provides new opportunities for sharing
- Visualizing compound data structures provides a quick window to assess whether or not sharing is occurring

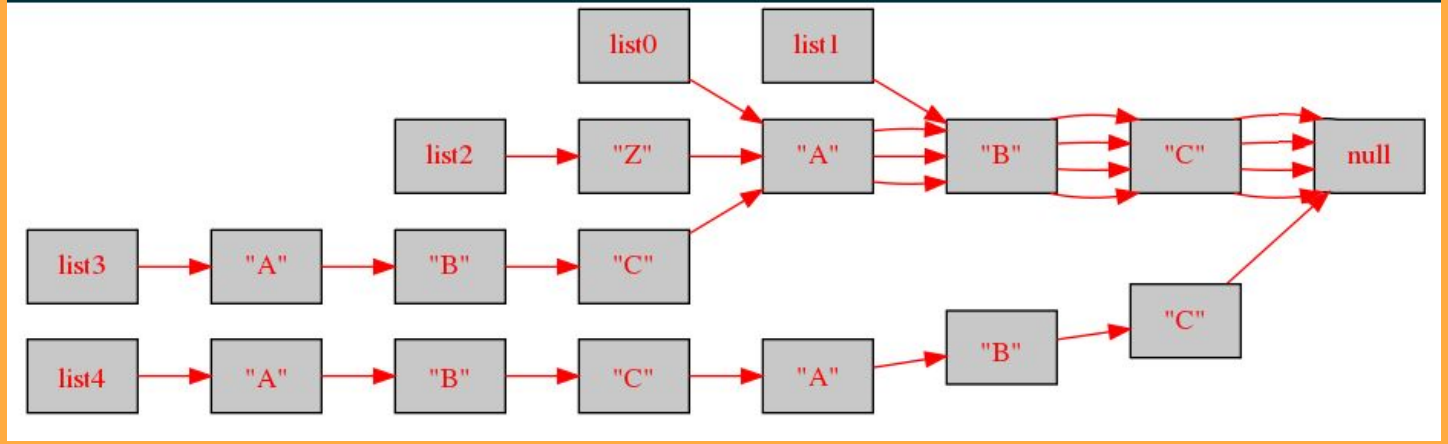
ShareVis : Visualizing Sharing

```haskell

import SpyShare

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)]
```



ShareVis : Visualizing Sharing

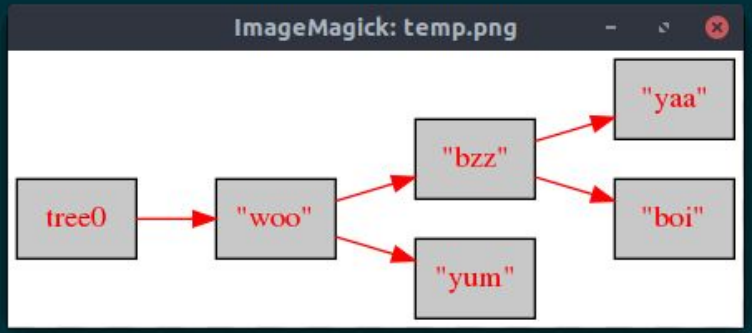
```haskell
import SpyShare

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)]
```



ShareVis : Conclusions

- Great for demonstrating simple substructural sharing
- Not 100% there for visualizing sharing in more complex data types like **VList**; requires some manual setup

Deep Lifting : An Example

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


deepBang :: Var Int -> Var Int -> Var Int
deepBang va vb = (apply2 (mkVarT +)
                         (apply (mkVarT foo) va)
                         (apply (mkVarT bar) vb))
```

Deep Lifting : Syntax Rewriting

Simple, in principle.

Recurse on the AST, performing the following replacements:

Base case: wrap atoms (literals, non-parameter variables)

x ⟹ (**mkVarT** x)

6 ⟹ (**mkVarT** 6)

Recursive case: Replace applications with apply

(*app* func args …) ⟹ (**apply** func args …)

Deep Lifting in Haskell

Two Candidate Approaches

- Template Haskell
  - 😎 *Powerful but* 😢 *Heavyweight*
  - 😢 *Designed more for code generation than transformation*
  - 😢 *Operates on surface syntax*

- Rewriting Rules
  - 😎 *Simple semantics: replacement rules in standard syntax*
  - 😎 *Can operate after rewriting to simpler Haskell Core syntax*
  - 😢 *Can only rewrite applications*
  - 😢 *Requires complex coordination with in-liner*
  - 😢 *Rules are applied non-deterministically*

Deep Lifting with Rewrite Rules

The initial example presented deep lifting at the definition level. With rewrite rules, we must lift at the application site. This requires inlining `bang`.

Before inlining:

```
shallowBang va vb =
    (apply2 (mkVarT bang) va vb)
```

After inlining:

```
shallowBang va vb =
    (apply2 (mkVarT (λ a b → (+) (foo a) (bar b)) va vb)
```

Deep Lifting with Rewrite Rules

So, we needed to design a system of rewrite rules to accomplish the desired lifting.

Problem one: Every intermediate step must be well-typed. This means a step can't just replace an ***app*** with an **apply**

Problem two: We can't specify whether the rules are applied top-down, bottom-up, right-to-left, or left-to right. Different orders of application must be *confluent*

Deep Lifting : The Rules

Deploy Scaffolding

```
(apply2 (mkVarT (λ x y → (F A))) vx vy) ⟹
(apply (apply2 (mkVarT (λ x y → F)) vx vy)
       (apply2 (mkVarT (λ x y → A)) vx vy))
```

Cleanup Scaffolding

```
(apply2 (mkVarT (λ x y → x)) vx vy) ⟹ vx
(apply2 (mkVarT (λ x y → y)) vx vy) ⟹ vy
(apply2 (mkVarT (λ x y → A)) vx vy) ⟹ (mkVarT A)
```

Deep Lifting with Rewrite Rules

Problem: These rules are sort of complicated. Do they make sense? Does order matter?

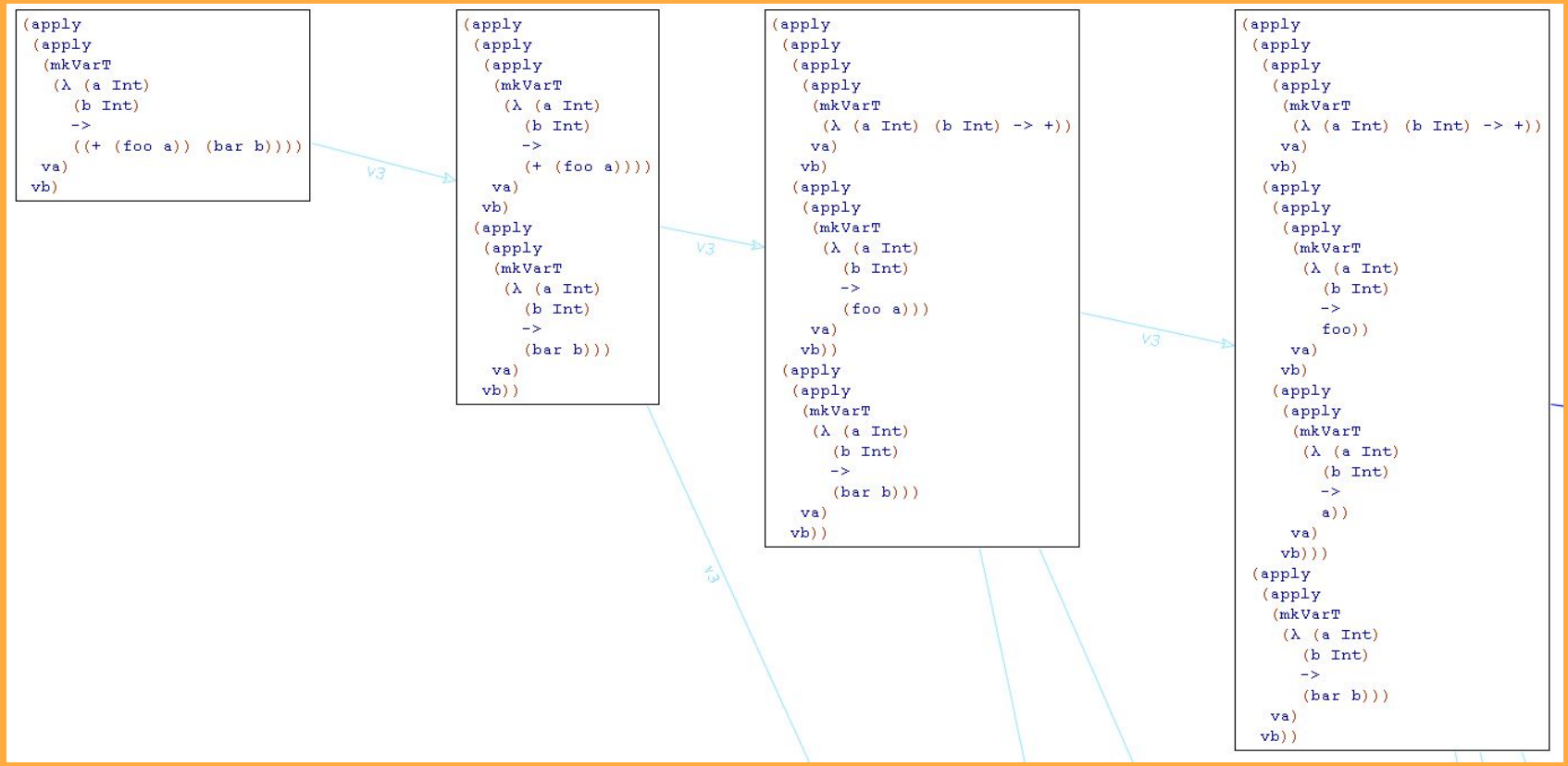
Problem: Testing the rules live in Haskell requires combing through massive raw compiler output

Solution: Simulate the rules in **PLT Redex**

Redex : A semantic engineering toolkit

- **Specify the syntax for a language**
  (in our case, a simply lambda-calculus-like language resembling Haskell Core)
- **Specify some reduction rules**
  (our deep lifting rewrite rules)
- **Specify some expressions to reduce**
  (our Bang example)

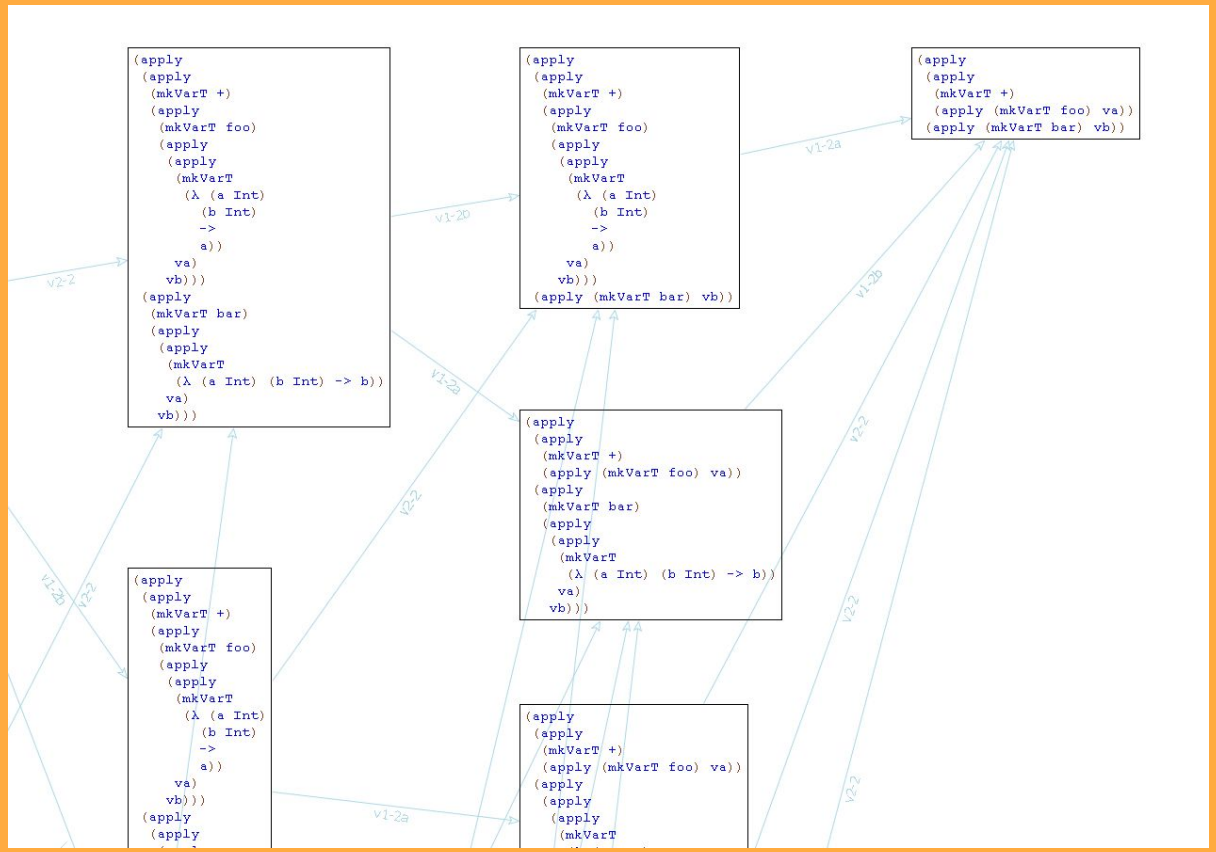
Modeling Deep Lifting in Redex



Deep Lifting

- Bang; shallow (upper left)
- Bang; deep (upper right)
- 56 intermediates (vert. truncated) along different paths of reduction

Modeling Deep Lifting in Redex



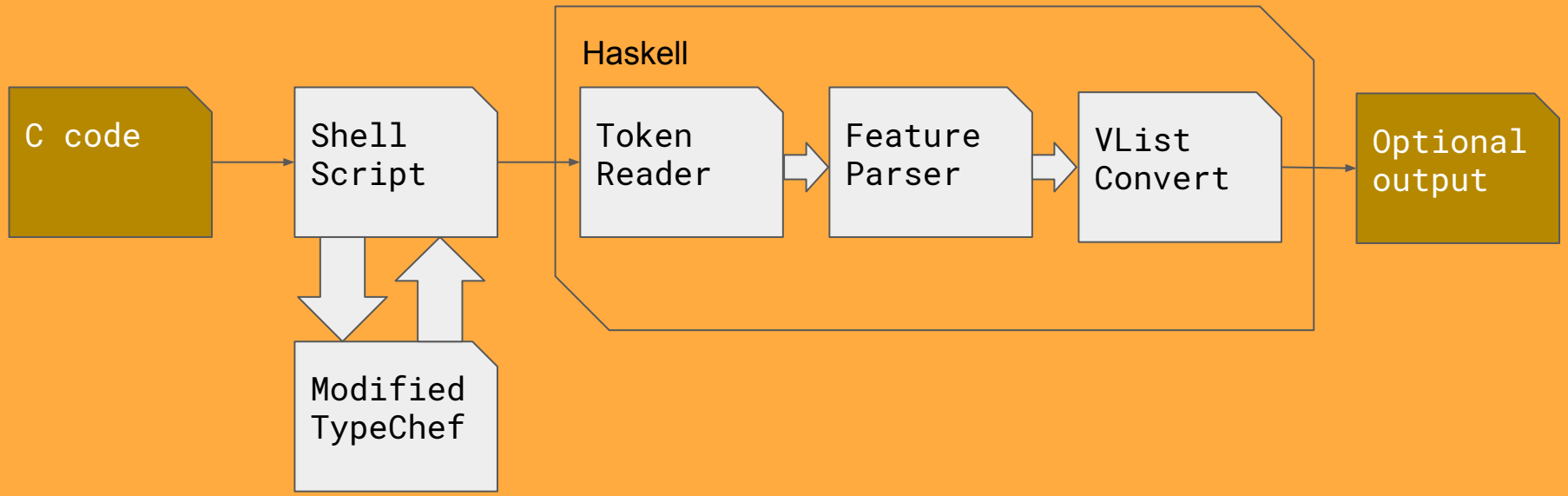
Deep Lifting: Conclusions

- Not yet a working product
- Workable in principle, but may depend on brittle coordination between compiler components
- Verbose GHC dumps (1MB+ even for simple programs) makes debugging the implementation time-consuming

Testing Harness

- We need a way to test brute-force against shallowly-lifted (and eventually deeply-lifted) analyses on real-world programs
- Specifically, we'll target C code annotated with preprocessor directives/macros
- To this end, we leveraged TypeChef, an existing suite of variability-aware C processing tools which includes a variability-aware preprocessor

Testing Harness



Personal Takeaways

- Reproducibility: Commit often, screenshot often, and **always save REPL and terminal sessions**
- Exploration: Extract value from large codebases by working backwards
- Ask questions early and often

Thanks!

- to everyone, for listening
- to Marsha, for supervising and suggesting me for this project
- to Ramy, for being a great advisor and for his general advice on academia and employment