Improving Data Structures

Visualization and Specialization

Don Stewart | wg2.8 | 2009-06-09

Functional Programming

Haiku or Karate?

Motivation

Despite what you might suspect

Haskell programs are not *yet* always the fastest **and** shortest

GHC is smart, but not **sufficiently smart**

<shootout>

galois

What can we do about it?

- Still more optimizations
  - Fusion (complexity changing!)
  - Constructor specialization
  - Domain-specific rewrite rules
- More and better parallelism
- Smarter runtime (constructor tag bits)
- Special purpose libraries
  - Data.Bytestring, Data.Binary

What about regular data types?

- Custom types and optimizations are great, but lots of programs use standard polymorphic data types
- Is there anything we can improve there?

- We need a nice way to look at how data structures are *actually* represented

A secret primop: unpackClosure#

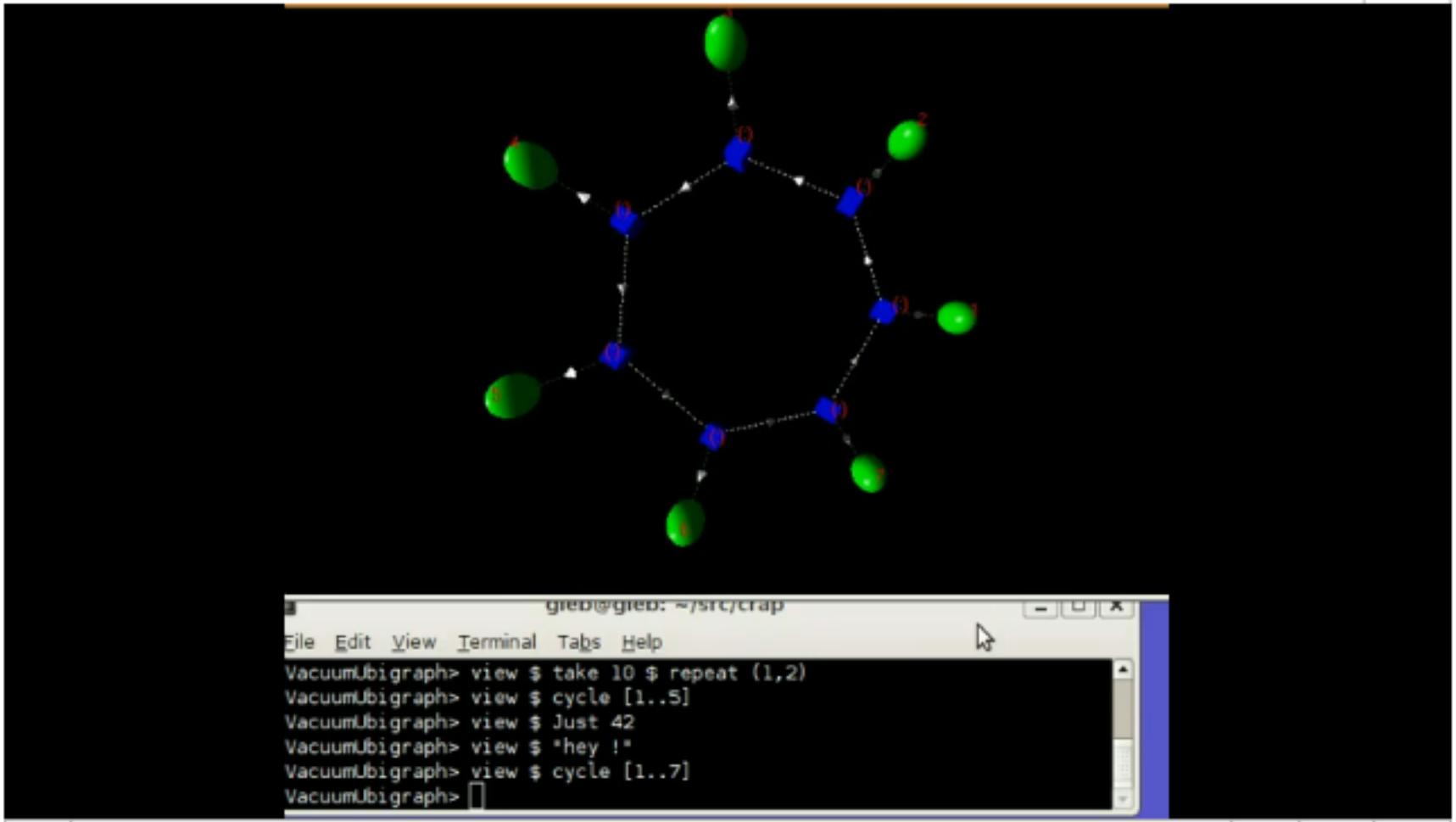
unpackClosure# :: a → (# Addr#, Array# b, ByteArray# #)

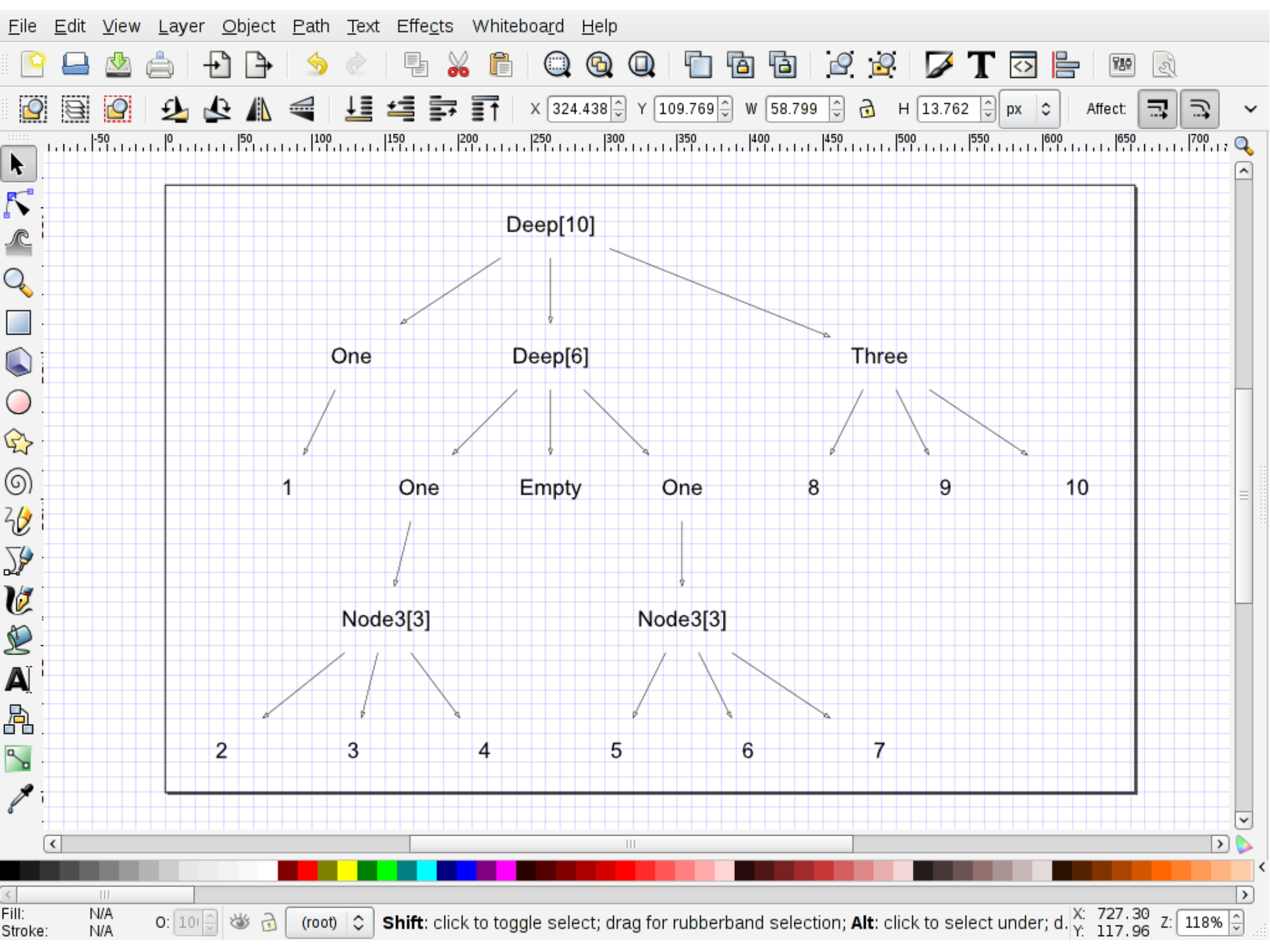
- Added for the GHCi Debugger
- Can write lots of interesting things with it:
  - sizeOfClosure :: a → Int
  - hasUnboxedFields :: a → Bool
  - view :: a → Graph
- Smuggling runtime reflection into Haskell

Everyday data types

<vacuum + cairo>

Vacuum + Ubigraph



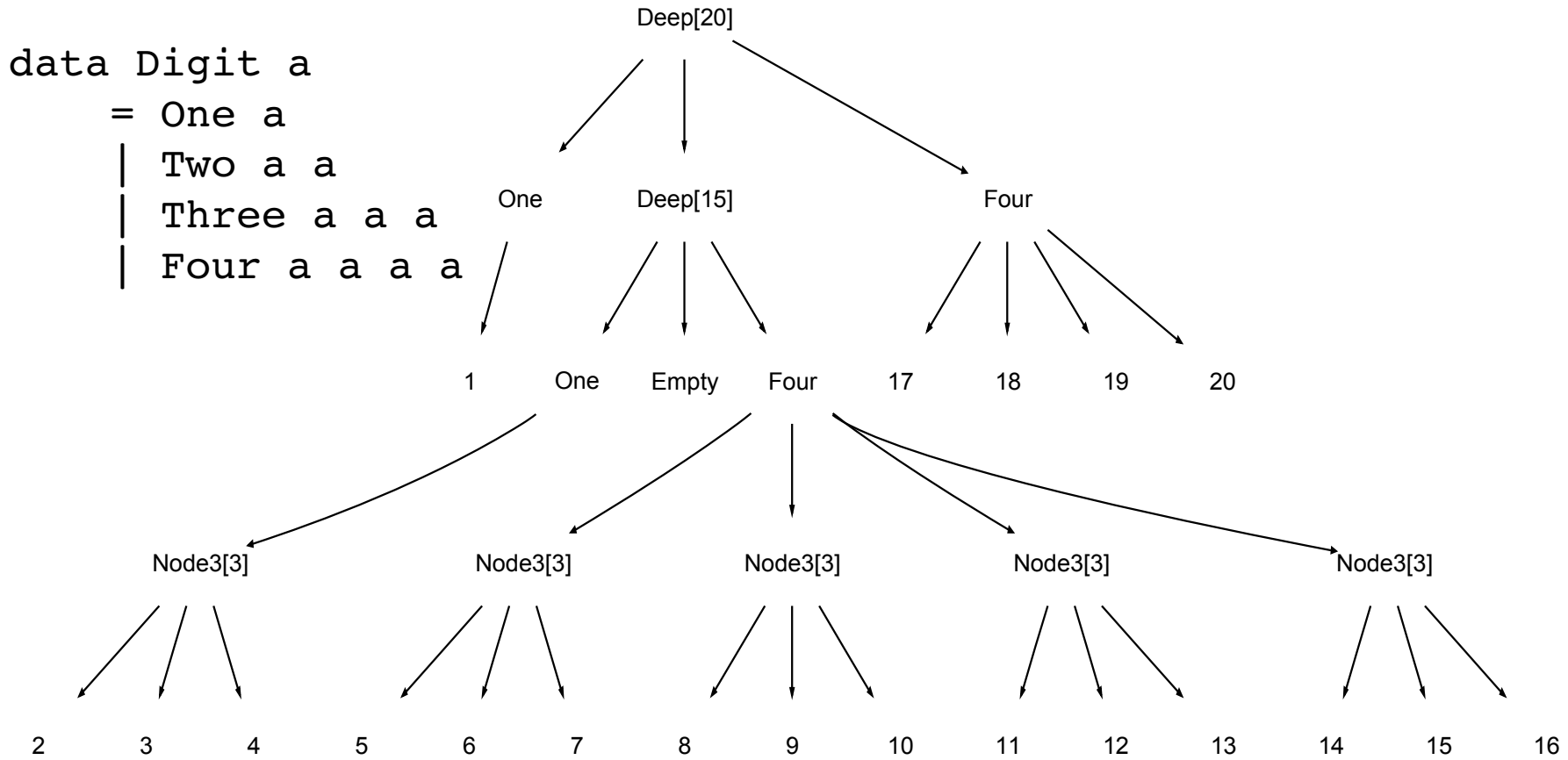

File Edit View Layer Object Path Text Effects Whiteboard Help
X 324.438 Y 109.769 W 58.799 H 13.762 px Affect:
Deep[10]
One
Deep[6]
Three
1
One
Empty
One
8
9
10
Node3[3]
Node3[3]
2
3
4
5
6
7
Fill: N/A
Stroke: N/A
O: 100
(root)
Shift: click to toggle select; drag for rubberband selection; Alt: click to select under; d.
X: 727.30
Y: 117.96
Z: 118%

```
data FingerTree a
    = Empty
    | Single a
    | Deep !Int
           !(Digit a)
           (FingerTree (Node a))
           !(Digit a)
```

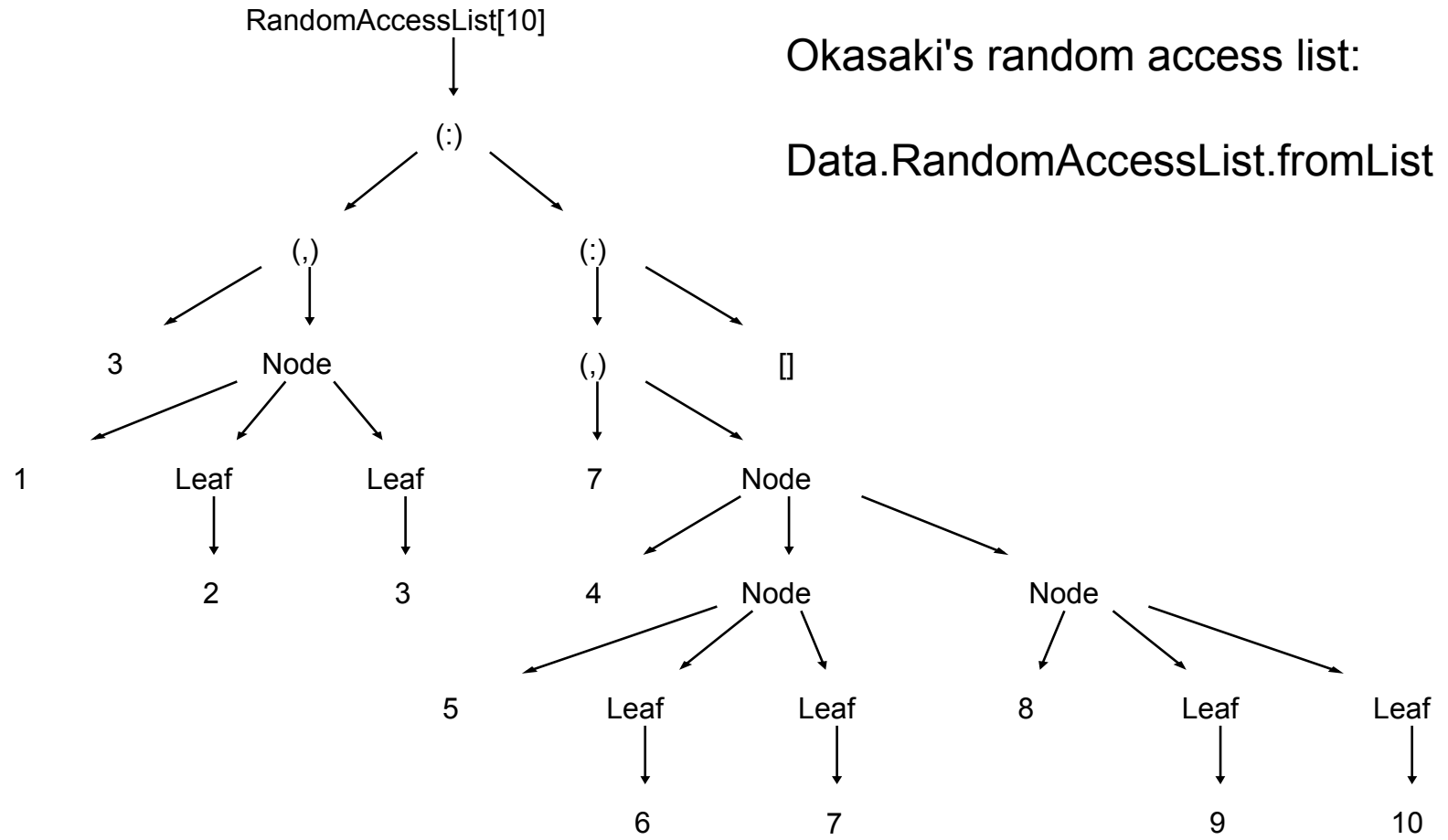
```
data Digit a
    = One a
    | Two a a
    | Three a a a
    | Four a a a a
```

Hinze & Patterson's finger tree

Data.Sequence.fromList [1..20]



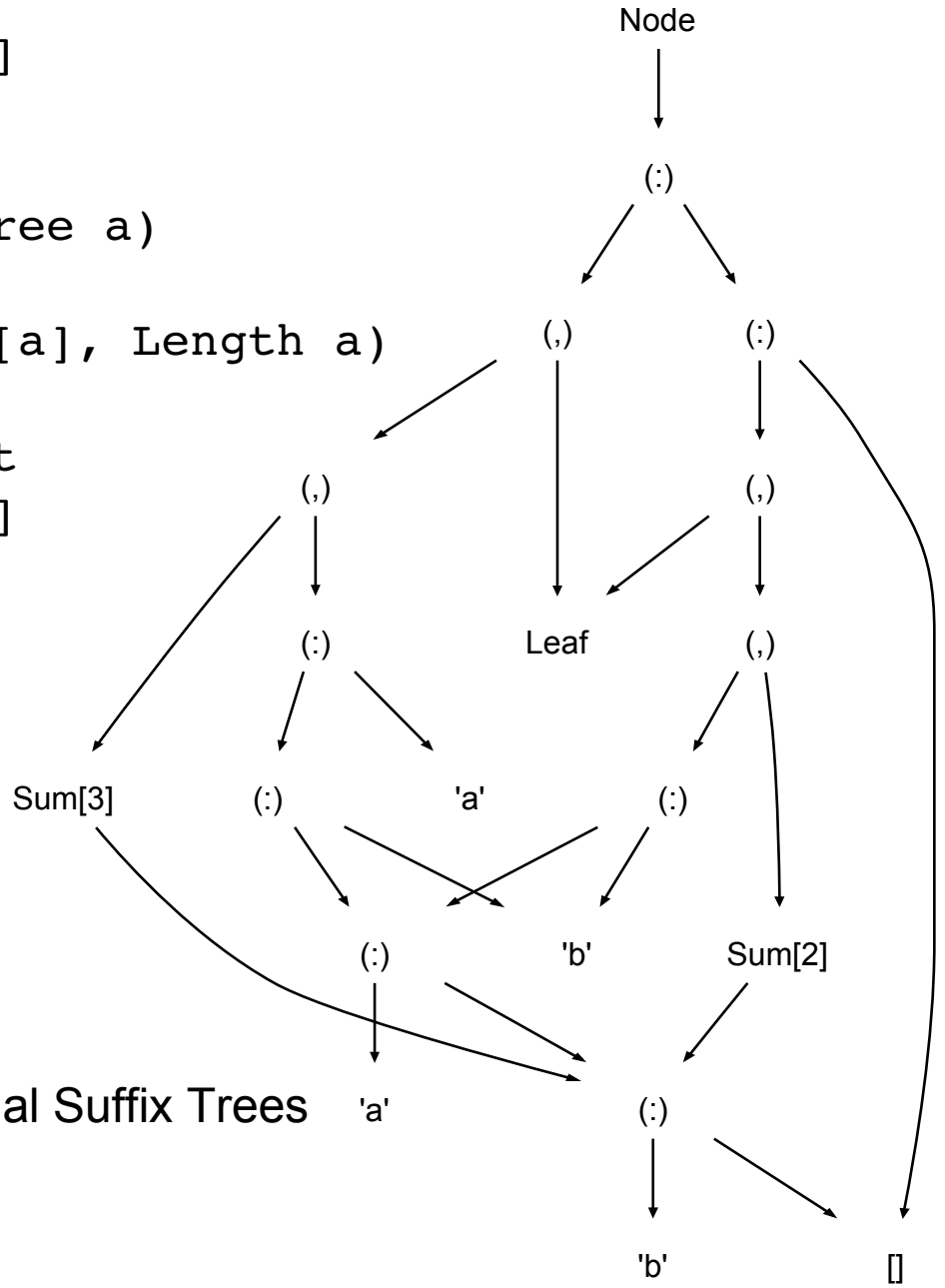
```
data RandomAccessList a
    = RandomAccessList !Int [(Int, CBTree a)]

data CBTree a
    = Leaf a
    | Node a !(CBTree a) !(CBTree a)
```



Okasaki's random access list:

Data.RandomAccessList.fromList [1..10]

```
data STree a = Node [Edge a]
             | Leaf

type Edge a = (Prefix a, STree a)

newtype Prefix a = Prefix ([a], Length a)

data Length a = Exactly !Int
              | Sum !Int [a]
```



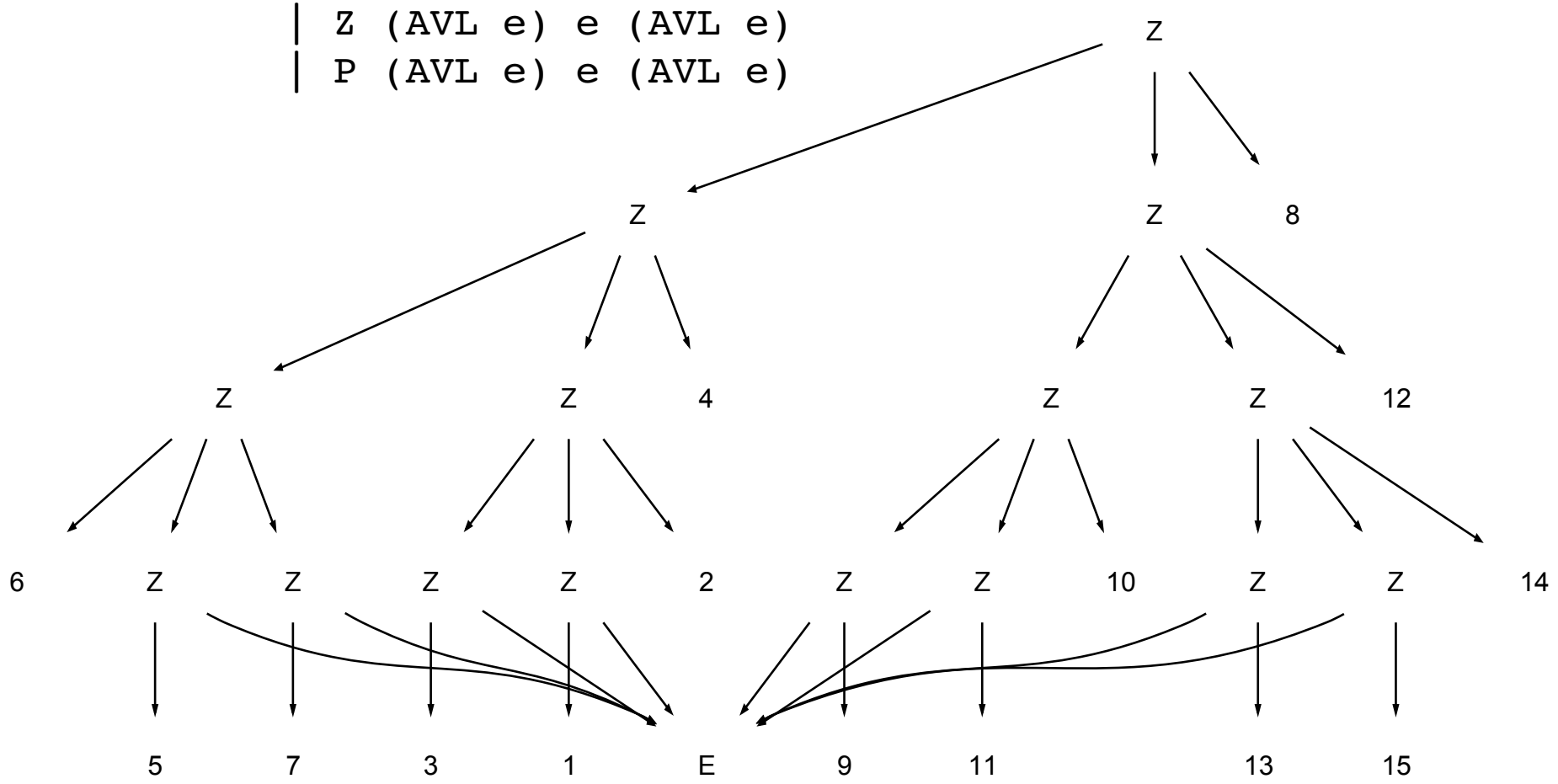
Giegerich and Kurtz's Purely Functional Suffix Trees

Data.SuffixTree.construct “abab”

Hey's AVL tree library

Data.Tree.AVL.asTreeL [1..15]

```
data AVL e = E
           | N (AVL e) e (AVL e)
           | Z (AVL e) e (AVL e)
           | P (AVL e) e (AVL e)
```



An Opportunity!

- Lots of parametrically polymorphic container types in common use
- All with (slow!) uniform representation

```
data Maybe a = Nothing | Just a
```

- But we “know” the type of 'a' statically!

```
readInt :: ByteString → Maybe Int
```

- How much **faster** do things get if we could specialize data types at each use!?

Challenge

Make parametrically polymorphic structures as efficient as monomorphic ones, when used at a known type

Remove the uniform representation penalty

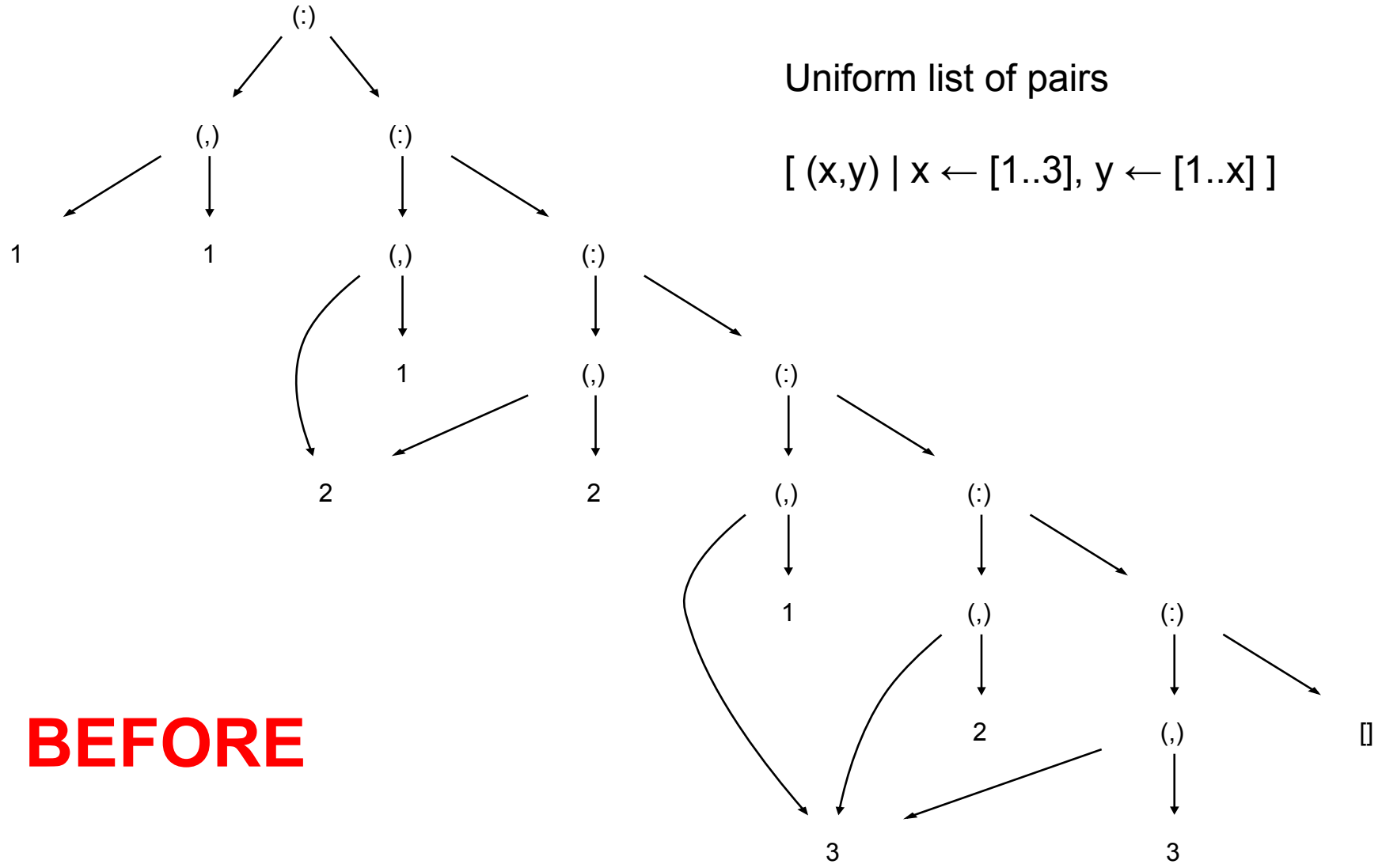
Inspiration: Habit and DPH

- Data Parallel Haskell
  - List-like interface
  - Radical restructuring under the hood (flattening)
- Habit: PSU/Galois “Systems Haskell”
  - Per-constructor representation annotations
- Whole-program compilers (a la JHC)
  - GHC Inliner is sort of there already...

Goals: uniform improvements for polymorphic containers

- Specialize polymorphic containers for each element type
- Retain a user interface of regular parametric polymorphism
  - Libraries should be mostly unchanged
- Set up uniform rules for specialization
  - Happy to sacrifice laziness for speed
- Open extn.: allow ad hoc representations

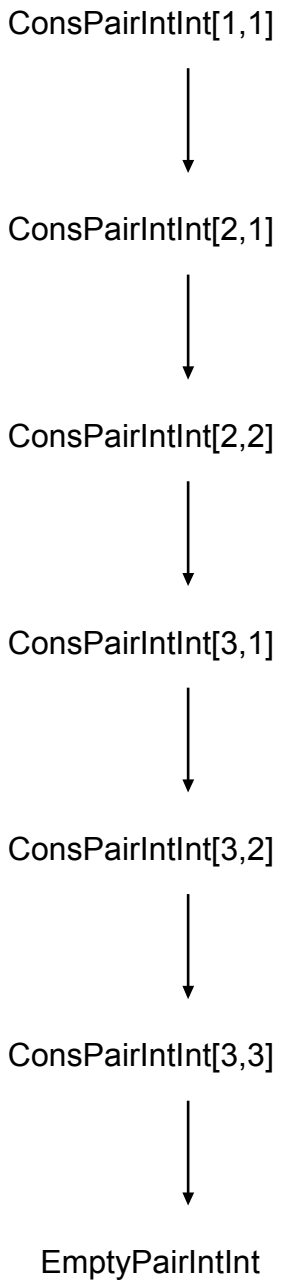
No more uniform representations!



Uniform list of pairs

[(x,y) | x ← [1..3], y ← [1..x]]

BEFORE



Specialized [(Int,Int)]

fromList [pair x y | x ← [1..3], y ← [1..x]]

AFTER :)

Mechanism is here!

- Type classes
  - Make decisions on a per-type basis
  - Open
  - Ad hoc
- Class-associated data types
  - Per-instance actual data types
  - Representation types added separately to function on those types

Self-optimizing tuples

```
{-# LANGUAGE TypeFamilies, MultiParamTypeClasses #-}

-- data (,) a b = (a, b)
class AdaptPair a b where

  data Pair a b          -- no representation yet

  curry   :: (Pair a b -> c) -> a -> b -> c
  fst     :: Pair a b -> a
  snd     :: Pair a b -> b
```


galois

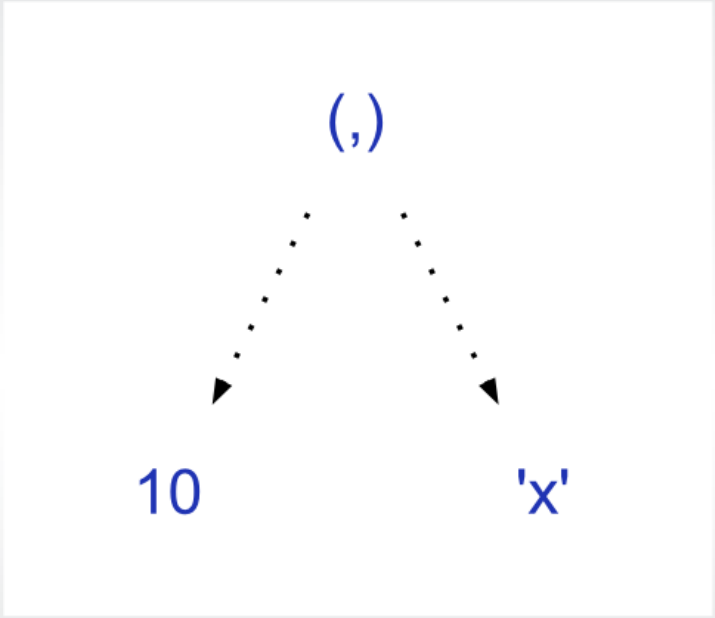
Functions on adaptive types

```
-- uncurry :: (a -> b -> c) -> ((a, b) -> c)
-- uncurry f p = f (fst p) (snd p)

uncurry :: AdaptPair a b
        => (a -> b -> c) -> (Pair a b -> c)
uncurry f p = f (fst p) (snd p)

pair    :: AdaptPair a b
        => a -> b -> Pair a b
pair    = curry id
```

Plugging in the representation type

```
instance AdaptPair Int Double where
    -- We can use our unpacking tricks per type
    data Pair Int Double
        = PairIntDouble {-# UNPACK #-}!Int
                        {-# UNPACK #-}!Double
    -- boilerplate views
    fst (PairIntDouble a _) = a
    snd (PairIntDouble _ b) = b
    curry f x y = f (PairIntDouble x y)
```

Non-uniform representations



PairIntChar[10,120]

galois

Joy

- Greater data density!
- More strictness info for common types
  - More CPR possible!
  - Should remove heap checks
- Interacts well with other optimizations
  - Fusion
- Can use ad hoc representation decisions

Careful... smart compiler wanted

- Don't want any dictionaries left over
  - GHC already *OK* at removing them
- Will rely heavily on inlining
  - Fake whole program compiler
- Need a lot of instances
  - Increases compile times...
- Library functions as templates, not directly reused (inlined, then specialized)

Creating instances

- Need instances for all combination of common element types
  - SYB generics to derive them
  - Template Haskell now supports unpacking pragmas and associated data types
- GHC gets a sore head when I enumerate all element types as instances

Lists

```
class AdaptList a where

    data List a

    empty   :: List a
    cons    :: a -> List a -> List a
    null    :: List a -> Bool
    head    :: List a -> a
    tail    :: List a -> List a
```


galois

List functions

```
fromList :: AdaptList a => [a] -> List a
fromList []     = empty
fromList (x:xs) = x `cons` fromList xs

(++) :: AdaptList a
     => List a -> List a -> List a
(++) xs ys
    | null xs   = ys
    | otherwise = head xs `cons` tail xs ++ ys
```

List functions

```
map :: (AdaptList a, AdaptList b)
    => (a -> b) -> List a -> List b

map f as = go as
 where
  go xs
   | null xs   = empty
   | otherwise = f (head xs) `cons` go (tail xs)
```


galois

List instances

```
instance AdaptList Int where
    data List Int
       = EmptyInt
       | ConsInt {-# UNPACK #-}!Int (List Int)

    empty = EmptyInt
    cons  = ConsInt
    null EmptyInt = True
    null _        = False
    head EmptyInt      = errorEmptyList "head"
    head (ConsInt x _) = x
```

Performance (preliminary)

- AdaptList a *vs* [a]
  - Pipelines of list functions: 15 – 30% faster
  - GC: 15 – 40% less allocation
  - More unboxing
- Need to be sure to have reliable dictionary removal

Data.List.sum

```
sum :: (AdaptList a, Num a) => List a -> a

sum l = go l 0
  where
    go xs !a
      | null xs   = a
      | otherwise = go (tail xs) (a + head xs)
```


galois

Data.List.sum

Use at *List Int* type:

```
$wgo :: List Int -> Int# -> Int#
$wgo xs n = case xs of
       EmptyInt      -> n
       ConsInt x xs -> $wgo xs (+# n x)
```

No unpacking. No views. No dictionaries. Elements aready in unboxed form!

galois

Trees and Sets

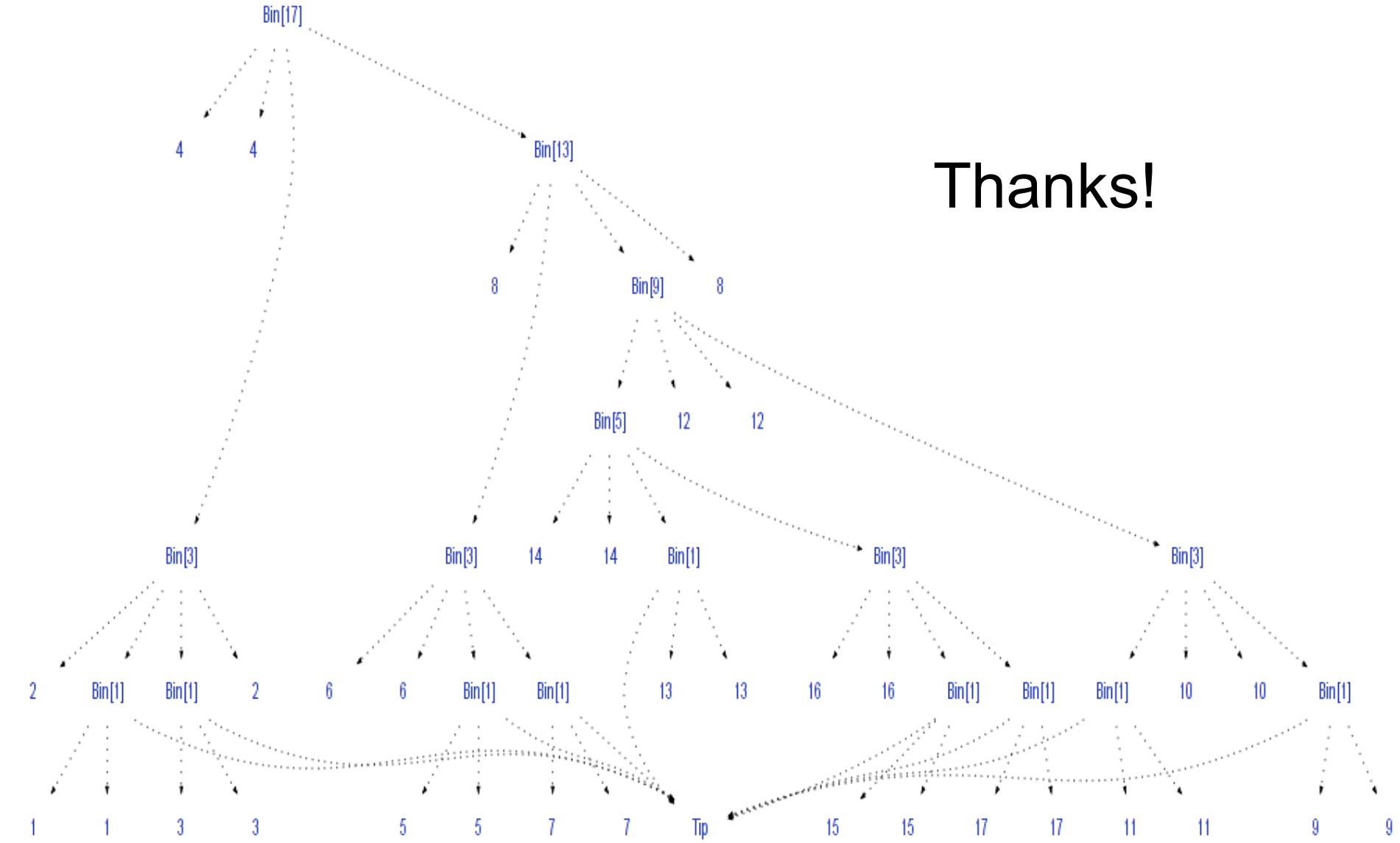
- Unpacking element types: obvious now
- Other ad hoc representation changes
  - Coalescing nodes in trees
    - Experiments: strong increase in density
  - Non-representation of singletons
  - Bools to bits

Non-uniform polymorphic containers

- So it is possible... generic approach to faster polymorphic structures
  - **Can we do this automatically?**
  - Rules based on strictness information?
- CPR enabled, can we do it on sum types?
- Treats inliner as poor man's whole program optimizer
- Nice: compiler extensions in the language

Where is all this?

- On Hackage
  - `cabal install vacuum-cairo`
  - `cabal install adaptive-containers`
- Other structures appearing (finger trees)
- BTW:
  - `cabal install fingertree`
  - `cabal install random-access-list`
  - `cabal install suffixtree`
  - `cabal install avltree`


Bin[17]
4
4
Bin[13]
8
Bin[9]
8
Bin[5]
12
12
Bin[3]
Bin[3]
14
14
Bin[1]
Bin[3]
Bin[3]
2
Bin[1]
Bin[1]
2
6
6
Bin[1]
Bin[1]
13
13
16
16
Bin[1]
Bin[1]
Bin[1]
10
10
Bin[1]
1
1
3
3
5
5
7
7
Tip
15
15
17
17
11
11
9
9

Thanks!