Persistent Data Structures in Haskell

Sebastiaan Visser

Example, user database

User datatype containing email and password.

```
type Email    = String
```

```
type Password = String
```

```
data User     = User
              { email    :: Email
              , password :: Password
              }
```

User database as mapping from email to user.

```
type UserDB = Map Email User                -- from Data.Map
```

Example, performing a signup

Add a new user to a database, the email address must be unique.

```
signup :: UserDB → User → Either String UserDB
```

```
signup db user =
  let mail = email user in
  case M.lookup mail db of
    Nothing → Right (M.insert mail user db)
    Just _  → Left "email in use"
```

Map as binary tree

Haskell's `Data.Map` is implemented as a size balanced binary tree.

Definition of a simplified binary tree, without key.

```
data Tree v = Leaf | Branch
                     { value :: v
                     , left  :: Tree v
                     , right :: Tree v
                     }
```

```
type Map k v = Tree (k, v)                 -- possible definition
```

Lookup on binary tree

```
lookup :: Ord k ⇒ k → Tree (k, v) → Maybe v
```

```
lookup k (Branch (w, v) l r) =
  case k `compare` w of
           LT → lookup k l
           EQ → Just v
           GT → lookup k r
lookup _ Leaf = Nothing
```

Insertion into binary tree

```haskell
insert :: Ord v ⇒ v → Tree v → Tree v
```

```haskell
insert v (Branch w l r) =
  case v `compare` w of
    LT → Branch w (insert v l) r
    EQ → Branch v l r
    GT → Branch w l (insert v r)
insert v Leaf = Branch v Leaf Leaf
```

Example, web application

Get post data out of web environment and try signup.

```
signupHandler :: TVar UserDB → Web String
```

```
signupHandler dbVar =
  do mail ← getPostVar "email"
     pass ← getPostVar "password"
     atomically $
       do db ← readTVar dbVar
          case signup db (User mail pass) of
            Left  err → return ("failed: " ++ err)
            Right db' → do writeTVar dbVar db'
                           return "signup ok"
```

The problem

When the programs terminates all data is lost.

The solution

Save the data structure on disk.

1. Fixed point annotations.
2. Morphisms and algebras.
3. File based storage heap.
4. Persistent recursive data structures.

Fixed Point Annotations

Original Tree definition

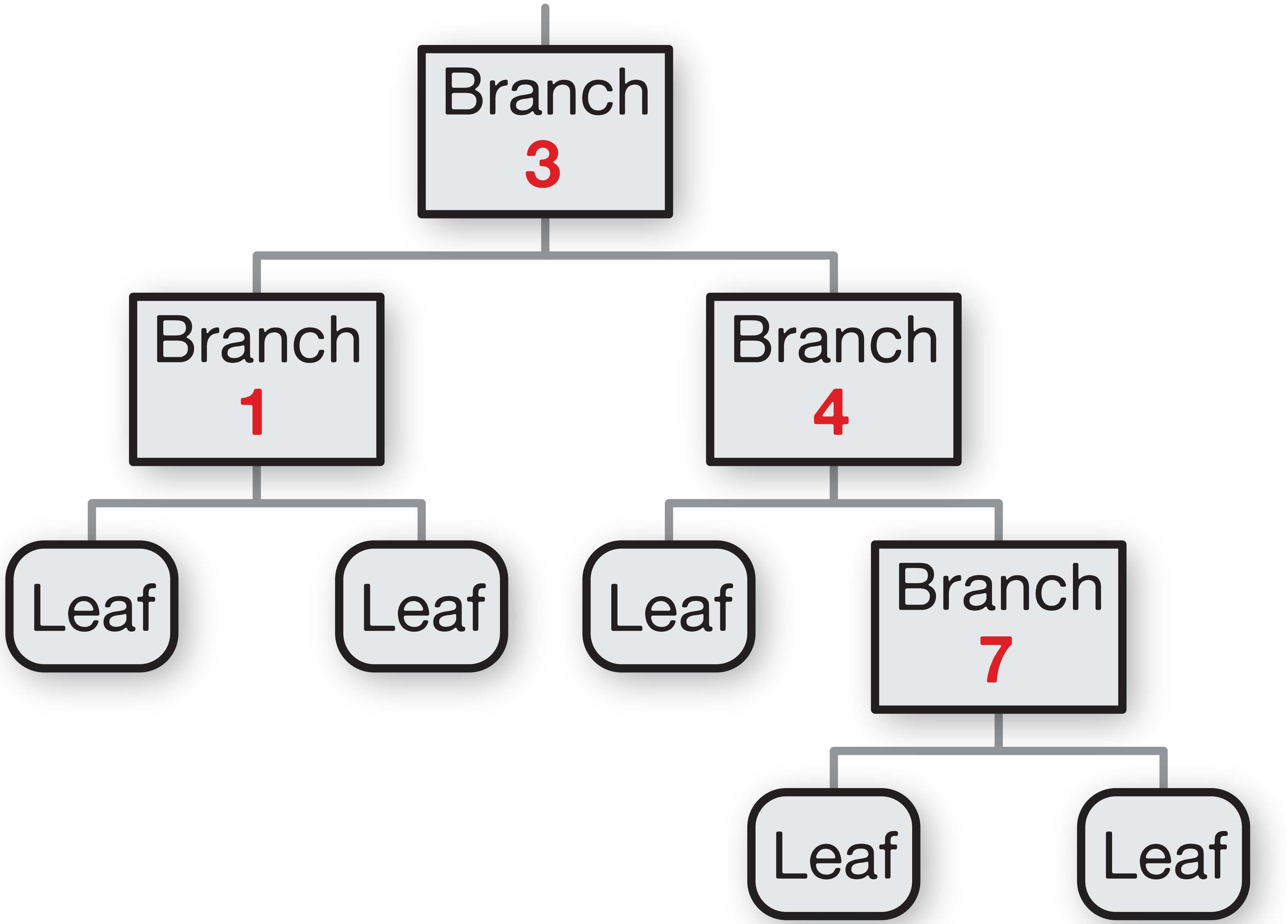
Recall the `Tree` datatype.

```
data Tree v = Leaf | Branch v (Tree v) (Tree v)
```

Example tree.

```
myTree :: Tree Int
```

```
myTree =
  Branch 3 (Branch 1 Leaf Leaf)
           (Branch 4 Leaf (Branch 7 Leaf Leaf))
```


Branch
3
Branch
1
Branch
4
Leaf
Leaf
Leaf
Branch
7
Leaf
Leaf

Fixed point combinator

Type level fixed point combinator.

```
newtype Fix (f :: * → *) = In { out :: f (Fix f) }
```

The fixed point combinator is sometimes called μ .

Open recursive definition

Parametrized with additional type variable for recursive position.

```
data TreeF v f = Leaf | Branch v f f
  deriving ( Eq, Ord, Show
           , Functor, Foldable, Traversable    -- ghc-6.12
           )
```

Fixed point combinator can be used to tie the knot.

```
type Tree v = Fix (TreeF v)
```

Smart constructors

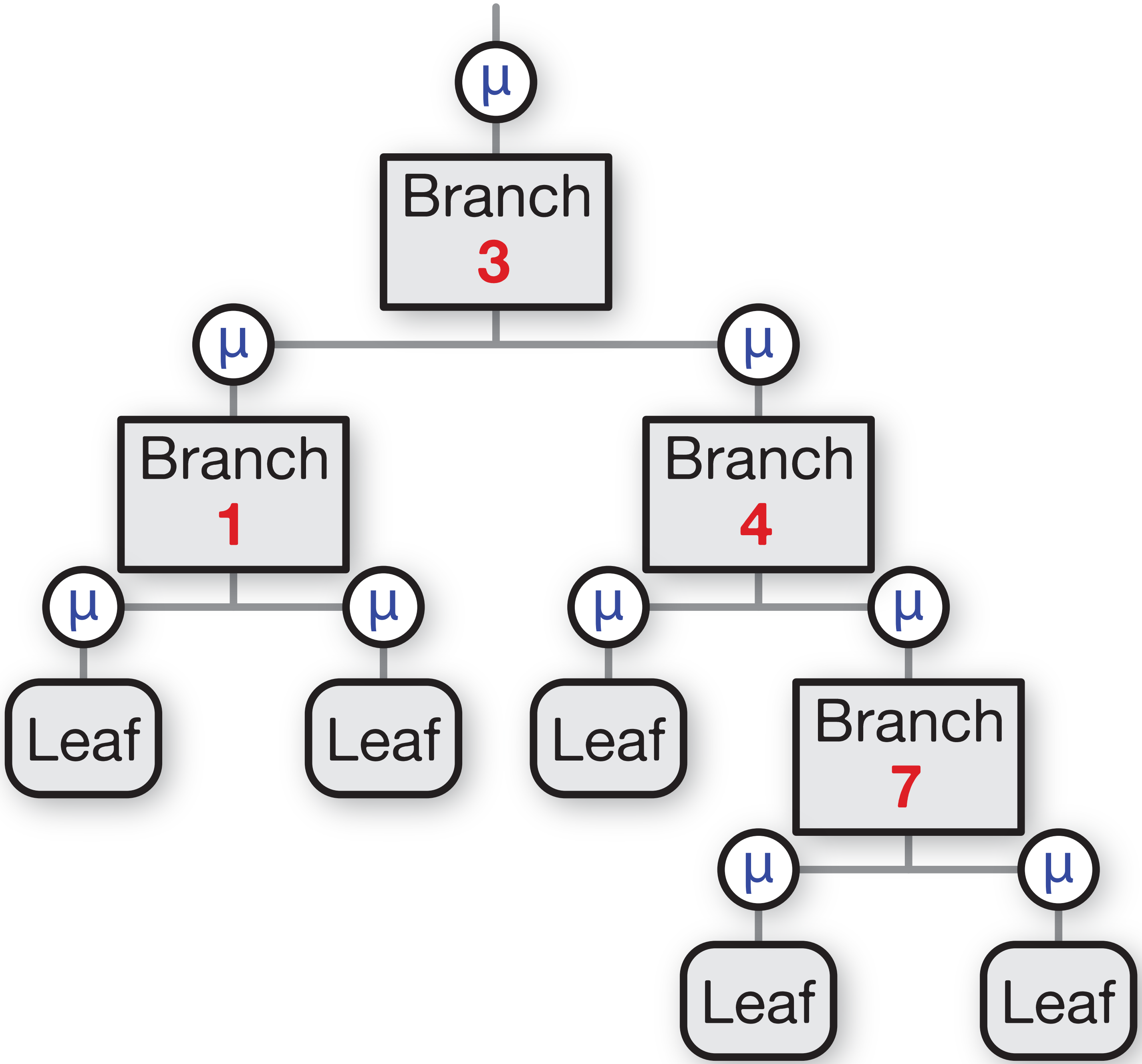
Smart constructors.

```haskell
leaf :: Tree v                              -- Tree v ≡ Fix (TreeF v)
leaf = In Leaf

branch :: v → Tree v → Tree v → Tree v
branch v l r = In (Branch v l r)
```

Example tree.

```haskell
myTree :: Tree Int
myTree =
  branch 3 (branch 1 leaf leaf)
           (branch 4 leaf (branch 7 leaf leaf))
```


μ
Branch
3
μ
μ
Branch
1
Branch
4
μ
μ
μ
μ
Leaf
Leaf
Leaf
Branch
7
μ
μ
Leaf
Leaf

Annotated fixed point combinator

Annotated fixed point stores additional annotation variable.

```
data FixA a f =
  InA { outa :: a f (FixA a f) }
```

Annotatation type classes

Unwrap a single node from annotation.

```
class Out a f m where
  outA :: FixA a f → m (f (FixA a f))
```

Wrap a single node in a fresh annotation.

```
class In a f m where
  inA :: f (FixA a f) → m (FixA a f)
```

Debug trace annotation

```
newtype Debug f a = D (f a)
```

Debug trace annotation

Annotation instances.

```
instance (Functor f, Show (f ())) ⇒ Out Debug f IO
   where outA (InA (D f)) =
            print ("Out", unit f) >> return f
```

```
instance (Functor f, Show (f ())) ⇒ In Debug f IO
   where inA f =
            print ("In", unit f) >> return (InA (D f))
```

```
unit :: Functor f ⇒ f a → f ()
unit = fmap (const ())
```

Smart constructors

Annotated tree.

```
type TreeA a v = FixA a (TreeF v)
```

Smart constructors.

```
leafA :: In a (TreeF v) m ⇒ m (TreeA a v)
```

```
leafA = inA Leaf
```

```
branchA :: In a (TreeF v) m
        ⇒ v → TreeA a v → TreeA a v → m (TreeA a v)
```

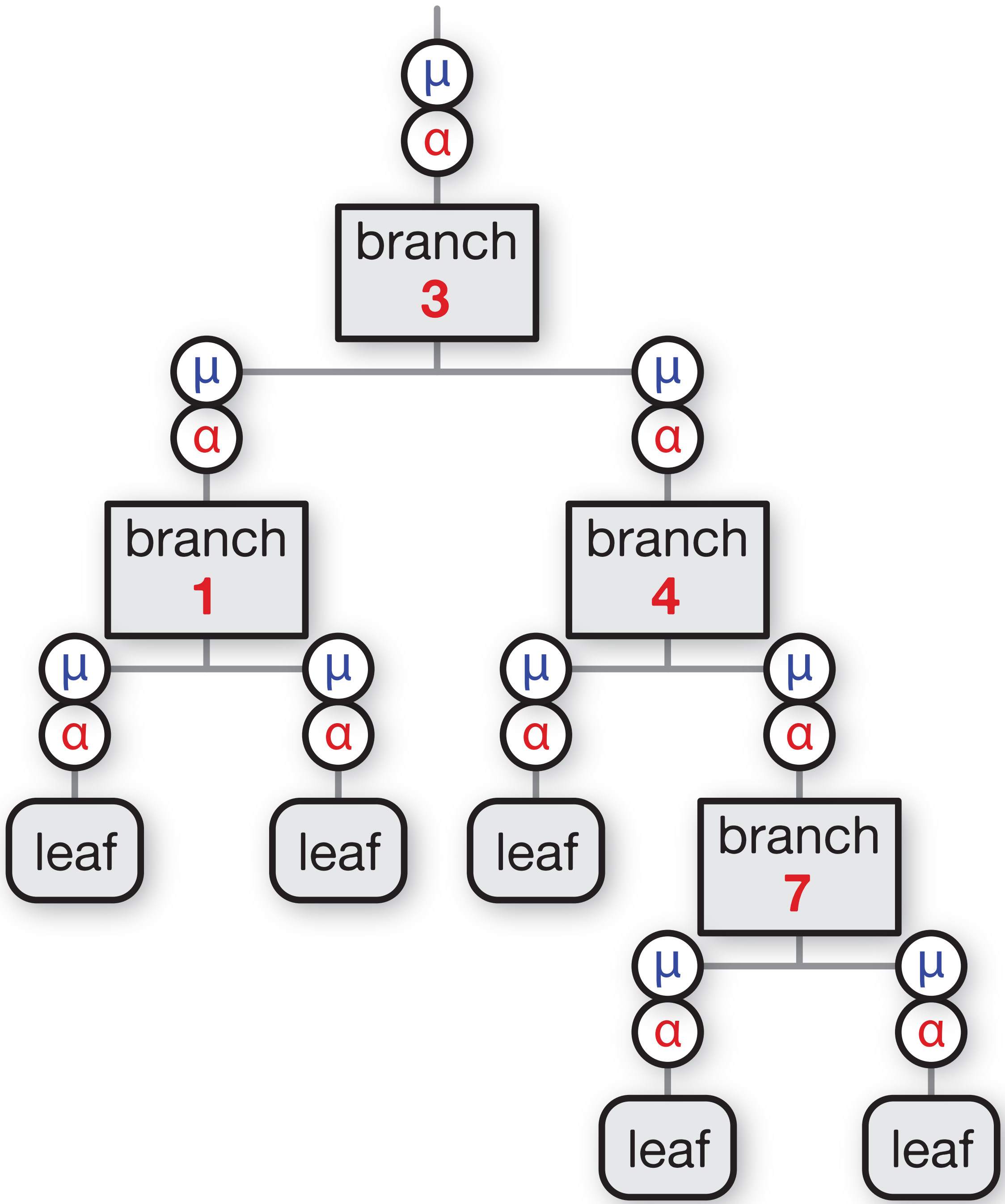
```
branchA v l r = inA (Branch v l r)
```

Annotated binary tree

Smart constructors.

```
myTreeA :: (Monad m, In a (TreeF Int) m)
        ⇒ m (TreeA a Int)
```

```
myTreeA =
  do l ← leafA
     d ← branchA 7 l l
     e ← branchA 1 l l
     f ← branchA 4 d l
     branchA 3 e f
```


μ
α
branch
3
μ
α
μ
α
branch
1
branch
4
μ
α
μ
α
μ
α
μ
α
leaf
leaf
leaf
branch
7
μ
α
μ
α
leaf
leaf

Annotated binary tree

Smart constructors.

```
myTreeD :: IO (TreeA Debug Int)
```

```
myTreeD = myTreeA
```

```
ghci> myTreeD
("In",Leaf)
("In",Branch 7 () ())
("In",Branch 1 () ())
("In",Branch 4 () ())
("In",Branch 3 () ())
{D (Branch 3 {D (Branch 1 {D Leaf} ...
```

Morphisms and Algebras

Abstracting away from recursion

Writing operations on annotated structures is hard.

Touching the recursive positions requires wrapping/unwrapping.

Algebraic operations

We will abstract away from recursion using **morphisms**.

Anamorphism

Like Haskell's `unfold`.

Coalgebra type:

```
type Coalg s f = s → f s
```

Corecursive anamorphic traversal:

```
anaA :: (In a f m, Monad m, Traversable f)
     ⇒ Coalg s f → s → m (FixA a f)
```

```
anaA coalg = inA <=< mapM (anaA coalg) . coalg
```

Binary tree from list.

```
fromListCoalg :: Coalg [v] (TreeF v)
```

```
fromListCoalg []     = Leaf
fromListCoalg (y:ys) =
  let l = take (length ys `div` 2) ys
      r = drop (length l         ) ys
  in Branch y l r
```

```
fromListA :: (Monad m, In a (TreeF v) m)
          ⇒ [v] → m (TreeA a v)
```

```
fromListA = anaA fromListCoalg
```

```haskell
squares :: IO (TreeA Debug (Int, Int))
```

```haskell
squares = fromListA [(1,1),(2,4),(3,9)]
```

```
ghci> squares
("In",Leaf)
("In",Leaf)
("In",Branch (2,4) () ())
("In",Leaf)
("In",Leaf)
("In",Branch (3,9) () ())
("In",Branch (1,1) () ())
{D (Branch (1,1) {D (Branch (2,4) {D Leaf} {D Leaf})}
                 {D (Branch (3,9) {D Leaf} {D Leaf})})}
```

Catamorphism

Like Haskell's `fold`.

Algebra type:

```
type Alg f r = f r → r
```

Recursive catamorphic traversal:

```
cataA :: (Monad m, Functor m, Out a f m, Traversable f)
      ⇒ Alg f r → FixA a f → m r
```

```
cataA alg = return . alg <=< mapM (cataA alg) <=< outA
```

Lookup on binary tree

```
lookupAlg :: Ord k ⇒ k → Alg (TreeF (k, v)) (Maybe v)
```

```
lookupAlg k (Branch (w, v) l r) =
  case k `compare` w of
            LT → l
            EQ → Just v
            GT → r
lookupAlg _ Leaf = Nothing
```

```
lookupA :: ( Monad m, Functor m
           , Ord k, Out a (TreeF (k, v)) m)
        ⇒ k → TreeA a (k, v) → m (Maybe v)
```

```
lookupA k = cataA (lookupAlg k)
```

```
ghci> squares >>= lookupA 3
("In",Leaf)
("In",Leaf)
("In",Branch (2,4) () ())
("In",Leaf)
("In",Leaf)
("In",Branch (3,9) () ())
("In",Branch (1,1) () ())
("Out",Branch (1,1) () ())
("Out",Branch (2,4) () ())
("Out",Leaf)
("Out",Leaf)
("Out",Branch (3,9) () ())
("Out",Leaf)
("Out",Leaf)
Just 9
```

Storage Heap

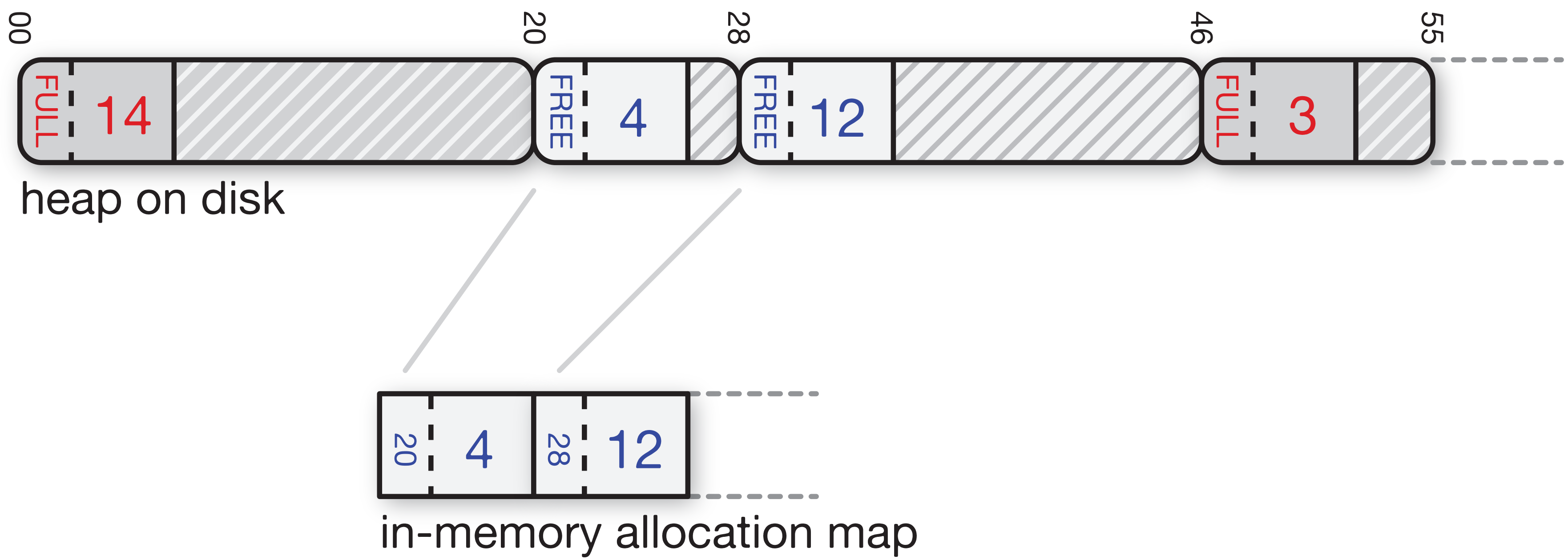
File based storage heap

Heap as a linear list of blocks of binary data.

A single block contains:

- `1 byte` used/free flag.
- `4 byte` payload byte size.
- `n byte` payload as binary stream.

Heap uses an in-memory map to perform allocation/freeing.



00
20
28
46
55
FULL
14
FREE
4
FREE
12
FULL
3
heap on disk
20
4
28
12
in-memory allocation map

Pointer type

A `Pointer` as byte offset into file.

```
type Offset = Integer
type Size   = Integer
```

```
newtype Pointer a = Ptr Offset
```

Heap context

The `Heap` context stores a file handle and an allocation map.

```
type AllocMap = Map Offset Size
```

```
newtype Heap a = Heap ( ReaderT Handle
                        ( StateT (Size, AllocMap)
                          IO )
                          a )
```

Heap operations

Basic operations:

```
read     :: Binary a ⇒ Pointer a → Heap a
write    :: Binary a ⇒ a         → Heap (Pointer a)
allocate ::            Integer   → Heap (Pointer a)
free     ::            Pointer a → Heap ()
```

Running a heap computation.

```
run      :: FilePath → Heap a    → IO ()
```

Binary type class

Serialize to binary, deserialize from binary:

```
class Binary t where
  put :: t → Put
  get :: Get t
```

From Hackage: binary + regular-extras or multirec-binary.

```
import Data.Binary
import Generics.Regular.Functions.Binary
import Generics.MultiRec.Binary
```

Persistent Data Structures

Pointer annotation

Wrapped pointer annotation.

```
newtype P f a = P { unP :: Pointer (f a) }
```

Unwrapping means reading from heap.

```
instance Binary (f (FixA P f)) ⇒ Out P f Heap
   where outA (InA (P f)) = read f
```

Wrapping means writing to heap.

```
instance Binary (f (FixA P f)) ⇒ In P f Heap
   where inA f = InA . P <$> write f
```

Persistent operations

```
type PersistentTree k = FixA P (TreeF k)
```

Build a tree on disk.

```
fromListP :: [v] → Heap (PersistentTree v)
```

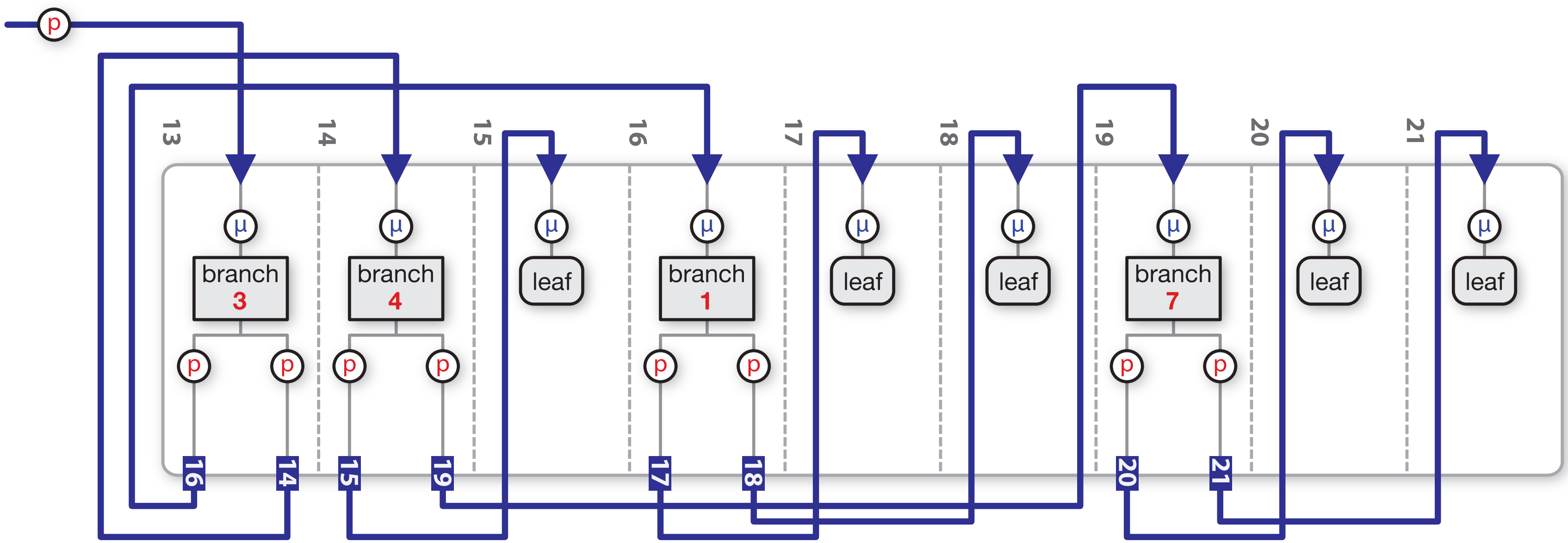
```
fromListP = fromListA
```

Lookup value from tree on disk.

```
lookupP :: Ord k ⇒ k
        → PersistentTree (k, v) → Heap (Maybe v)
```

```
lookupP = lookupA
```

```
fromListP [3,1,4,7]
```



Creating square database

BuildSquareDB.hs

```
main :: IO ()
```

```
main =
  do run "squares.db" $
       do p ← fromListP (map (\a → (a, a*a)) [1..10])
          storeRootPtr (p :: FixA P (TreeF (Int, Int)))
     putStrLn "Database created."
```

```
storeRootPtr :: FixA P f → Heap ()
```

Looking up squares

LookupSquares.hs

```
main' :: IO ()
```

```
main' =
  run "squares.db" $ forever $
    do liftIO $ putStr "Give a number> "
       num ← Prelude.read <$> liftIO getLine
       sqr ← fetchRootPtr >>= lookupP num
       liftIO $ print ( num :: Int                -- actual lookup
                      , sqr :: Maybe Int
                      )
```

```
fetchRootPtr :: Heap (FixA P f)
```

```
$ ghc --make BuildSquareDB.hs -o build-squares-db
$ ghc --make LookupSquares.hs -o lookup-squares
...
$ ./build-square-db
Database created.
$ ls *.db
squares.db
$ hexdump squares.db
0000000 54 68 69 73 20 69 73 20 6a 75 73 74 20 61 20 66
0000010 61 6b 65 20 65 78 61 6d 70 6c 65 21 21 21 21 0a
...
$ ./lookup-squares
Give a number> 3
(3, Just 9)
Give a number> 9
(9, Just 81)
Give a number> 12
(12, Nothing)
^C
$ _
```

Conclusion

This frameworks allows you to:

- Write pure Haskell data structures.
- Generically annotate operations with I/O code.
- Save recursive data structures on disk.
- Allow incremental access to slices of data.

But unfortunately you have to:

- Abstract away from recursion using morphisms.
- Use the final operations in a monadic context.

In the thesis

- Data structure modification.
  (see backup slides)
- Applicative algebras.
- Regaining laziness in strict contexts.
- Persistent higher order recursive data types.
  (finger trees as GADTs)

Future work

- Make prototype into real library!
- Allow sharing (requires reference counting)
- Allow cycles (requires garbage collecting)
- Transactional in-memory cache.
- Incremental folds.
- ...

More

Thesis PDF:
  github.com/sebastiaanvisser/msc-thesis

Source code prototype:
  github.com/sebastiaanvisser/islay

How to build these slides:
  github.com/sebastiaanvisser/lhs2html5

Backup

Functor, Traversable

Functor uses plain function.

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Traversable uses effectful computation.

```
class Traversable f where                          -- simplified
  mapM :: Monad m ⇒ (a → m b) → f a → m (f b)
```

Comparison.

```
fmap ::              (a →   b) → f a →   f b
mapM :: Monad m ⇒ (a → m b) → f a → m (f b)
```

Modification

Modify a node inside an annotation.

```
class (Out a f m, In a f m, Monad m) ⇒ OutIn a f m where
  annIO :: (f (FixA a f) → m (f (FixA a f)))
        → FixA a f       → m (FixA a f)
```

Default implementation performs unwrap/wrap:

```
annIO f = inA <=< f <=< outA
```

Pointer instance.

```
instance Binary (f (FixA P f)) ⇒ OutIn P f Heap
   where annIO g (InA (P f)) =
               InA . P <$> (write =<< g =<< fetch f)
```

```
fetch :: Binary a ⇒ Pointer a → Heap a
```

Endomorphic Paramorphism

Algebra type:

```
type Endo f a = f (FixA a f, FixA a f) → FixA a f
```

Recursive endo-paramorphic traversal:

```
endoA :: (Functor m, OutIn a f m, Traversable f)
      ⇒ Endo f a → FixA a f → m (FixA a f)
```

Insert algebra:

```
insert :: Ord v ⇒ v → Endo (TreeF v) a
```

Allocate

Allocate scans the in-memory allocation map for a free block.

```
allocate :: Integer → Heap (Pointer a)
```

```
allocate size =
  do (end, unused) ← get                              -- from StateT
     case atLeast size unused of
       offset:_ → useBlockAt offset                    -- from map
       _        → useBlockAt end                       -- grow heap
  where atLeast s = M.elems . M.filter (>= s)
```

```
useBlockAt :: Integer → Heap (Pointer a)
```

Indexed datatypes

Higher order annotated fixed point.

```
data HFixA
    (a  :: ((* → *) → * → *) → (* → *) → * → *)
    (f  ::  (* → *) → * → *)
    (ix ::    *)
  = HInA { houta :: a f (HFixA a f) ix }
```

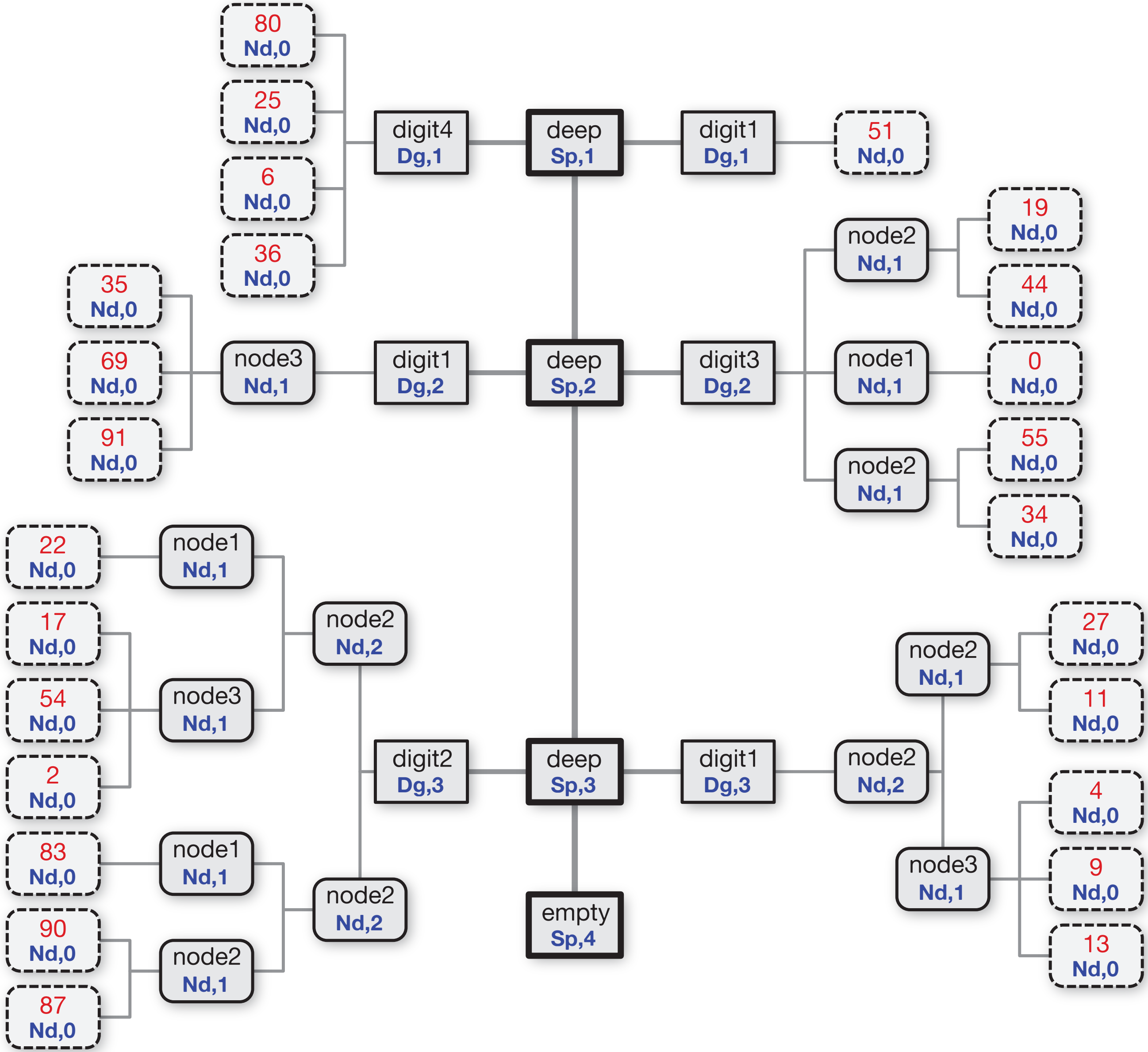
Higher order pointer annotation.

```
newtype HP
        (f  :: (* → *) → * → *)
        (b  ::   * → *)
        (ix ::   *)
      = HP { unHP :: Pointer (f b ix) }
```

```
data Sp ; data Dg ; data Nd
```

```
data FT (v :: *) (f :: * → *) :: * → * where
  Empty  ::                                 FT v f (Sp, S c)
  Single :: f (Dg, S c)                   → FT v f (Sp, S c)
  Deep   :: f (Dg, S c)
         → f (Sp, S (S c))
         → f (Dg, S c)                    → FT v f (Sp, S c)
  Digit1 :: f (Nd, c)                     → FT v f (Dg, S c)
  Digit2 :: f (Nd, c) → f (Nd, c)         → FT v f (Dg, S c)
  Digit3 :: f (Nd, c) → f (Nd, c)
         → f (Nd, c)                      → FT v f (Dg, S c)
  Digit4 :: f (Nd, c) → f (Nd, c)
         → f (Nd, c) → f (Nd, c)          → FT v f (Dg, S c)
  Node2  :: f (Nd, c) → f (Nd, c)         → FT v f (Nd, S c)
  Node3  :: f (Nd, c) → f (Nd, c)
         → f (Nd, c)                      → FT v f (Nd, S c)
  Value  :: v                             → FT v f (Nd, Z)
```

```
type FingerTreeP v = HFixA HP (FT v) (Sp, S Z)
```



80 Nd,0
25 Nd,0
6 Nd,0
36 Nd,0
digit4 Dg,1
deep Sp,1
digit1 Dg,1
51 Nd,0
19 Nd,0
44 Nd,0
node2 Nd,1
35 Nd,0
69 Nd,0
91 Nd,0
node3 Nd,1
digit1 Dg,2
deep Sp,2
digit3 Dg,2
node1 Nd,1
0 Nd,0
node2 Nd,1
55 Nd,0
34 Nd,0
22 Nd,0
node1 Nd,1
17 Nd,0
54 Nd,0
2 Nd,0
node3 Nd,1
node2 Nd,2
digit2 Dg,3
deep Sp,3
digit1 Dg,3
node2 Nd,2
node2 Nd,1
27 Nd,0
11 Nd,0
node3 Nd,1
4 Nd,0
9 Nd,0
13 Nd,0
83 Nd,0
node1 Nd,1
90 Nd,0
87 Nd,0
node2 Nd,1
node2 Nd,2
empty Sp,4

Example, authenticate a user

Does the database contain a user with the right email and password?

```
authenticate :: UserDB → User → Bool
```

```
authenticate db user =
  case M.lookup (email user) db of
    Just (User _ p) | p ≡ password user → True
    _                                   → False
```

More

Thesis PDF:
github.com/sebastiaanvisser/msc-thesis

Source code prototype:
github.com/sebastiaanvisser/islay

How to build these slides:
github.com/sebastiaanvisser/lhs2html5