Generic Storage in Haskell

WGP 2010

Sebastiaan Visser and Andres Löh
Utrecht University

September 26, 2010

Motivation

- Functional programmers naturally use data structures (such as finite maps) to maintain program data.
- Normal data structures are not persistent – at the end of a program session, all data is lost.
- Even if we serialize the whole data structure, we have to read/write the entire data structure at once and hold everything in memory in between.
- We could use a database, but then we have to convert between the Haskell data structure and the database's data model.

This talk

A generic framework for library writers to define persistent functional data structures.

Outline

- Datatypes as fixed points.
- Annotations and effects.
- Lifting operations to the annotated setting.
- A file-based storage heap.
- Persistent data structures.

Fixed points

Finite maps as binary trees

Similar to Haskell's Data.Map library:

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

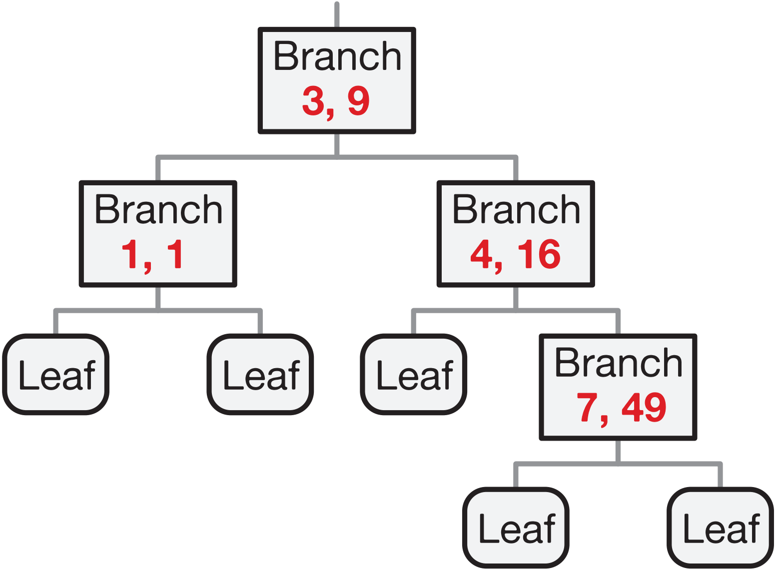
Finite maps as binary trees

Similar to Haskell's Data.Map library:

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

Running example:

```
myTree :: Tree Int Int
myTree = Branch 3 9 (Branch 1 1  Leaf
                                 Leaf)
                    (Branch 4 16 (Branch 7 49 Leaf
                                              Leaf)
                                 Leaf)
```



Branch
3, 9
Branch
1, 1
Branch
4, 16
Leaf
Leaf
Leaf
Branch
7, 49
Leaf
Leaf

Making the recursive structure explicit

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

Making the recursive structure explicit

```
data TreeF k v r = Leaf
                 | Branch k v r
                              r
  deriving Functor
```

Making the recursive structure explicit

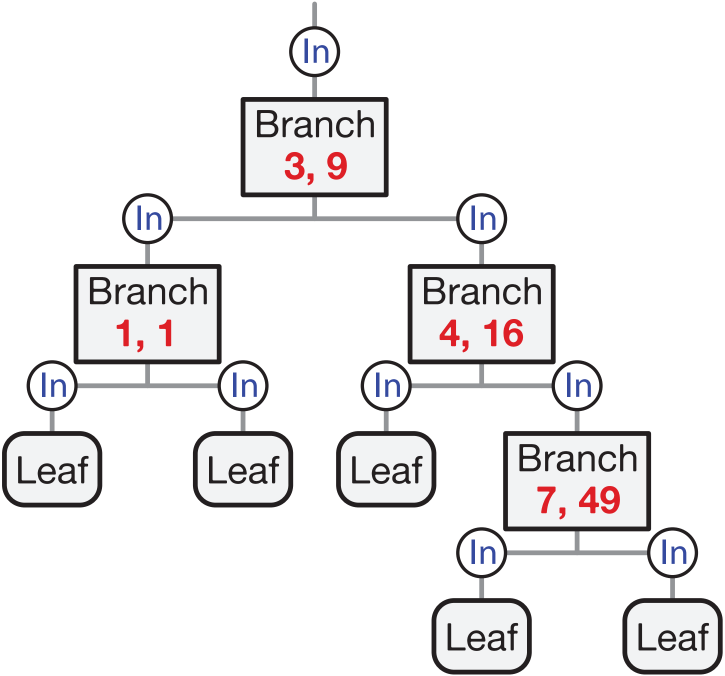
```
data Tree_F k v r = Leaf
                  | Branch k v r
                                r
  deriving Functor
newtype μ f  = In {out :: f (μ f)}
type Tree k v = μ (Tree_F k v)
```

Making the recursive structure explicit

```
data Tree_F k v r = Leaf
                  | Branch k v r
                                r
  deriving Functor
newtype μ f  = In {out :: f (μ f)}
type Tree k v = μ (Tree_F k v)
myTree  :: Tree Int Int
myTree  = Branch 3 9 (Branch 1 1   Leaf
                                   Leaf)
                     (Branch 4 16 (Branch 7 49 Leaf
                                               Leaf)
                                   Leaf)
```

Making the recursive structure explicit

```
data TreeF k v r = Leaf
                 | Branch k v r
                              r
  deriving Functor
newtype μ f  = In {out :: f (μ f)}
type Tree k v = μ (TreeF k v)
myTreef :: Tree Int Int
myTreef = branch 3 9 (branch 1 1  leaf
                                  leaf)
                     (branch 4 16 (branch 7 49 leaf
                                               leaf)
                              leaf)
leaf           = In Leaf
branch k v l r = In (Branch k v l r)
```


In
Branch
3, 9
In
In
Branch
1, 1
Branch
4, 16
In
In
In
In
Leaf
Leaf
Leaf
Branch
7, 49
In
In
Leaf
Leaf

Annotations

Annotated fixed points

"Normal" fixed point:

$$\textbf{newtype}\ \mu\ \mathsf{f} = \mathsf{In}\ \{\mathsf{out} :: \mathsf{f}\ (\mu\ \mathsf{f})\}$$

Annotated fixed points

"Normal" fixed point:

$$\textbf{newtype}\ \mu\ \mathsf{f} = \mathsf{In}\ \{\mathsf{out} :: \mathsf{f}\ (\mu\ \mathsf{f})\}$$

Annotated fixed point:

$$\textbf{type}\ \mu_\alpha\ \alpha\ \mathsf{f} = \mu\ (\alpha\ \mathsf{f})$$

Annotated fixed points

"Normal" fixed point:

newtype μ f = In {out :: f (μ f)}

Annotated fixed point:

type μ_α α f = μ (α f)

Identity annotation:

newtype Id f a = Id {unId :: f a}

Effectful annotations

We use annotations to attach effects to the folding and unfolding of the fixed-point combinator:

$$
\begin{aligned}
&\textbf{class}\ \mathsf{Monad}\ \mathsf{m} \Rightarrow \mathsf{In}\ \alpha\ \mathsf{f}\ \mathsf{m}\ \textbf{where} \\
&\quad \mathsf{in}_\alpha :: \mathsf{f}\ (\mu_\alpha\ \alpha\ \mathsf{f}) \rightarrow \mathsf{m}\ (\mu_\alpha\ \alpha\ \mathsf{f}) \\
&\textbf{class}\ \mathsf{Monad}\ \mathsf{m} \Rightarrow \mathsf{Out}\ \alpha\ \mathsf{f}\ \mathsf{m}\ \textbf{where} \\
&\quad \mathsf{out}_\alpha :: \mu_\alpha\ \alpha\ \mathsf{f} \rightarrow \mathsf{m}\ (\mathsf{f}\ (\mu_\alpha\ \alpha\ \mathsf{f}))
\end{aligned}
$$

Effectful annotations

We use annotations to attach effects to the folding and unfolding of the fixed-point combinator:

$$\textbf{class}\ \text{Monad m} \Rightarrow \text{In}\ \alpha\ \text{f m}\ \textbf{where}$$
$$\quad \text{in}_\alpha :: \text{f}\ (\mu_\alpha\ \alpha\ \text{f}) \to \text{m}\ (\mu_\alpha\ \alpha\ \text{f})$$
$$\textbf{class}\ \text{Monad m} \Rightarrow \text{Out}\ \alpha\ \text{f m}\ \textbf{where}$$
$$\quad \text{out}_\alpha :: \mu_\alpha\ \alpha\ \text{f} \to \text{m}\ (\text{f}\ (\mu_\alpha\ \alpha\ \text{f}))$$

The identity annotation has no effect:

$$\textbf{instance}\ \text{In Id f Identity}\ \textbf{where}$$
$$\quad \text{in}_\alpha = \text{return} \circ \text{In} \circ \text{Id}$$
$$\textbf{instance}\ \text{Out Id f Identity}\ \textbf{where}$$
$$\quad \text{out}_\alpha = \text{return} \circ \text{unId} \circ \text{out}$$

Debug trace annotation

Same type as the identity annotation:

$$\textbf{newtype}\ \text{Debug f a} = \text{D}\ \{\text{unD} :: \text{f a}\}$$

Debug trace annotation

Same type as the identity annotation:

```
newtype Debug f a = D {unD :: f a}
```

This time, we attach an IO effect:

```
instance (Functor f, Show (f ())) ⇒ In    Debug f IO where
  inα   f             = print ("In"  , units f) ≫ return (In (D f))
instance (Functor f, Show (f ())) ⇒ Out Debug f IO where
  outα (In (D f)) = print ("Out", units f) ≫ return f
```

The function units instantiates the recursive positions with units:

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

Building an annotated tree

Annotated binary trees:

$$\textbf{type}\ \mathsf{Tree}_{\alpha}\ \alpha\ \mathsf{k}\ \mathsf{v} = \mu_{\alpha}\ \alpha\ (\mathsf{Tree}_{\mathsf{F}}\ \mathsf{k}\ \mathsf{v})$$

Building an annotated tree

Annotated binary trees:

$$\textbf{type}\ \text{Tree}_\alpha\ \alpha\ \text{k}\ \text{v} = \mu_\alpha\ \alpha\ (\text{Tree}_\text{F}\ \text{k}\ \text{v})$$

Monadic, but polymorphic in the annotation:

$$
\begin{array}{l}
\text{myTree}_\alpha :: \text{In}\ \alpha\ (\text{Tree}_\text{F}\ \text{Int}\ \text{Int})\ \text{m} \Rightarrow \text{m}\ (\text{Tree}_\alpha\ \alpha\ \text{Int}\ \text{Int}) \\
\text{myTree}_\alpha = \\
\quad \textbf{do}\ \text{l} \leftarrow \text{leaf}_\alpha \\
\qquad \text{d} \leftarrow \text{branch}_\alpha\ 7\ 49\ \text{l}\ \ \text{l} \\
\qquad \text{e} \leftarrow \text{branch}_\alpha\ 1\ 1\ \ \ \text{l}\ \ \text{l} \\
\qquad \text{f} \leftarrow \text{branch}_\alpha\ 4\ 16\ \text{d}\ \text{l} \\
\qquad \text{branch}_\alpha\ 3\ 9\ \text{e}\ \text{f}
\end{array}
$$

$$
\begin{array}{ll}
\text{leaf}_\alpha & = \text{in}_\alpha\ \text{Leaf} \\
\text{branch}_\alpha\ \text{k}\ \text{v}\ \text{l}\ \text{r} & = \text{in}_\alpha\ (\text{Branch}\ \text{k}\ \text{v}\ \text{l}\ \text{r})
\end{array}
$$

Specializing to a particular annotation

$\mathsf{myTree_D} :: \mathsf{IO}\ (\mathsf{Tree}_\alpha\ \mathsf{Debug}\ \mathsf{Int}\ \mathsf{Int})$
$\mathsf{myTree_D} = \mathsf{myTree}_\alpha$

```
ghci> myTree_D
("in",Leaf)
("in",Branch 7 49 () ())
("in",Branch 1 1 () ())
("in",Branch 4 16 () ())
("in",Branch 3 9 () ())
{D (Branch 3 9 {D (Branch 1 1 {D Leaf} ...
```

Operations

Manipulating annotated trees

- Writing operations on annotated structures requires adding and removing annotations.
- If we do not pay attention, all the code becomes monadic and cluttered with maintaining the annotations.
- We therefore try to lift the recursion patterns, not the operations themselves.

Catamorphism

```
type Algebra f r = f r → r

cata :: Functor f ⇒ Algebra f r → μ f → r
cata φ = φ ∘ fmap (cata φ) ∘ out
```

Catamorphism

$$\textbf{type}\ \text{Algebra f r} = \text{f r} \rightarrow \text{r}$$

$$\text{cata} :: \text{Functor f} \Rightarrow \text{Algebra f r} \rightarrow \mu\ \text{f} \rightarrow \text{r}$$
$$\text{cata}\ \phi = \phi \circ \text{fmap}\ (\text{cata}\ \phi) \circ \text{out}$$

$$\text{lookup}_{\text{ALG}} :: \text{Ord k} \Rightarrow \text{k} \rightarrow \text{Algebra}\ (\text{Tree}_{\text{F}}\ \text{k v})\ (\text{Maybe v})$$
$$\text{lookup}_{\text{ALG}}\ \text{k Leaf} = \text{Nothing}$$
$$\text{lookup}_{\text{ALG}}\ \text{k}\ (\text{Branch n x l r}) = \textbf{case}\ \text{k `compare` n}\ \textbf{of}$$
$$\text{LT} \rightarrow \text{l}$$
$$\text{EQ} \rightarrow \text{Just x}$$
$$\text{GT} \rightarrow \text{r}$$

$$\text{lookup k} = \text{cata}\ (\text{lookup}_{\text{ALG}}\ \text{k})$$

Catamorphism

$\textbf{type}\ \mathsf{Algebra\ f\ r} = \mathsf{f\ r} \rightarrow \mathsf{r}$

$\mathsf{cata} :: \mathsf{Functor\ f} \Rightarrow \mathsf{Algebra\ f\ r} \rightarrow \mu\ \mathsf{f} \rightarrow \mathsf{r}$
$\mathsf{cata}\ \phi = \phi \circ \mathsf{fmap}\ (\mathsf{cata}\ \phi) \circ \mathsf{out}$

$\mathsf{lookup_{ALG}} :: \mathsf{Ord\ k} \Rightarrow \mathsf{k} \rightarrow \mathsf{Algebra}\ (\mathsf{Tree_F\ k\ v})\ (\mathsf{Maybe\ v})$
$\mathsf{lookup_{ALG}\ k\ Leaf} = \mathsf{Nothing}$
$\mathsf{lookup_{ALG}\ k}\ (\mathsf{Branch\ n\ x\ l\ r}) = \textbf{case}\ \mathsf{k}\ \text{`}\mathsf{compare}\text{`}\ \mathsf{n}\ \textbf{of}$
$\quad \mathsf{LT} \rightarrow \mathsf{l}$
$\quad \mathsf{EQ} \rightarrow \mathsf{Just\ x}$
$\quad \mathsf{GT} \rightarrow \mathsf{r}$

$\mathsf{lookup\ k} = \mathsf{cata}\ (\mathsf{lookup_{ALG}\ k})$

Example:

$\mathsf{lookup\ 4} \qquad \mathsf{myTree_f}$

Catamorphism

$$\textbf{type}\ \text{Algebra f r} = \text{f r} \rightarrow \text{r}$$

$$\text{cata} :: \text{Functor f} \Rightarrow \text{Algebra f r} \rightarrow \mu\ \text{f} \rightarrow \text{r}$$
$$\text{cata}\ \phi = \phi \circ \text{fmap}\ (\text{cata}\ \phi) \circ \text{out}$$

$$\text{lookup}_{\text{ALG}} :: \text{Ord k} \Rightarrow \text{k} \rightarrow \text{Algebra}\ (\text{Tree}_{\text{F}}\ \text{k v})\ (\text{Maybe v})$$
$$\text{lookup}_{\text{ALG}}\ \text{k Leaf} = \text{Nothing}$$
$$\text{lookup}_{\text{ALG}}\ \text{k}\ (\text{Branch n x l r}) = \textbf{case}\ \text{k `compare` n}\ \textbf{of}$$
$$\text{LT} \rightarrow \text{l}$$
$$\text{EQ} \rightarrow \text{Just x}$$
$$\text{GT} \rightarrow \text{r}$$

$$\text{lookup k} = \text{cata}\ (\text{lookup}_{\text{ALG}}\ \text{k})$$

Example:

$$\text{lookup}_{\text{ALG}}\ 4\ (\text{fmap}\ (\text{lookup}\ 4)\ (\text{out myTree}_{\text{f}}))$$

Catamorphism

$$\textbf{type}\ \text{Algebra}\ f\ r = f\ r \rightarrow r$$

$$\text{cata} :: \text{Functor}\ f \Rightarrow \text{Algebra}\ f\ r \rightarrow \mu\ f \rightarrow r$$
$$\text{cata}\ \phi = \phi \circ \text{fmap}\ (\text{cata}\ \phi) \circ \text{out}$$

```
lookup_ALG :: Ord k ⇒ k → Algebra (Tree_F k v) (Maybe v)
lookup_ALG k Leaf              = Nothing
lookup_ALG k (Branch n x l r) = case k `compare` n of
                                    LT → l
                                    EQ → Just x
                                    GT → r

lookup k = cata (lookup_ALG k)
```

Example:

$\text{lookup}_{\text{ALG}}\ 4\ (\text{fmap}\ (\text{lookup}\ 4)\ (\text{Branch}\ (3\ 9\ \ldots \qquad \ldots)))$

Catamorphism

type Algebra f r = f r → r

cata :: Functor f ⇒ Algebra f r → μ f → r
cata ϕ = ϕ ∘ fmap (cata ϕ) ∘ out

$\text{lookup}_{\text{ALG}}$ :: Ord k ⇒ k → Algebra (Tree_{F} k v) (Maybe v)
$\text{lookup}_{\text{ALG}}$ k Leaf = Nothing
$\text{lookup}_{\text{ALG}}$ k (Branch n x l r) = **case** k ‘compare‘ n **of**
  LT → l
  EQ → Just x
  GT → r

lookup k = cata ($\text{lookup}_{\text{ALG}}$ k)

Example:

$\text{lookup}_{\text{ALG}}$ 4 (Branch (3 9 Nothing (Just 16)))

Catamorphism

$$\textbf{type}\ \text{Algebra f r} = \text{f r} \to \text{r}$$

$$\text{cata} :: \text{Functor f} \Rightarrow \text{Algebra f r} \to \mu\ \text{f} \to \text{r}$$
$$\text{cata}\ \phi = \phi \circ \text{fmap}\ (\text{cata}\ \phi) \circ \text{out}$$

```
lookup_ALG :: Ord k => k -> Algebra (Tree_F k v) (Maybe v)
lookup_ALG k Leaf               = Nothing
lookup_ALG k (Branch n x l r) = case k `compare` n of
                                    LT -> l
                                    EQ -> Just x
                                    GT -> r

lookup k = cata (lookup_ALG k)
```

Example:

Just 16

Catamorphism with annotations

$$
\begin{aligned}
\mathsf{cata} \quad &:: \mathsf{Functor}\ \mathsf{f} \Rightarrow \\
&\quad \mathsf{Algebra}\ \mathsf{f}\ \mathsf{r} \to \mu\ \mathsf{f} \quad \to \mathsf{r} \\
\mathsf{cata} \quad \phi &= \quad \phi \circ \mathsf{fmap} \quad (\mathsf{cata} \quad \phi) \circ \mathsf{out}
\end{aligned}
$$

Catamorphism with annotations

$$\mathsf{cata}_\alpha :: (\mathsf{Out}\ \alpha\ \mathsf{f}\ \mathsf{m}, \mathsf{Traversable}\ \mathsf{f}) \Rightarrow$$
$$\mathsf{Algebra}\ \mathsf{f}\ \mathsf{r} \to \mu_\alpha\ \alpha\ \mathsf{f} \to \mathsf{m}\ \mathsf{r}$$
$$\mathsf{cata}_\alpha\ \phi = \mathsf{return} \circ \phi \triangleleft \mathsf{mapM}\ (\mathsf{cata}_\alpha\ \phi) \triangleleft \mathsf{out}_\alpha$$

$$(\triangleleft) \quad :: \mathsf{Monad}\ \mathsf{m} \Rightarrow (\mathsf{b} \to \mathsf{m}\ \mathsf{c}) \to (\mathsf{a} \to \mathsf{m}\ \mathsf{b}) \to \mathsf{a} \to \mathsf{m}\ \mathsf{c}$$
$$\mathsf{mapM} :: (\mathsf{Traversable}\ \mathsf{t}, \mathsf{Monad}\ \mathsf{m}) \Rightarrow (\mathsf{a} \to \mathsf{m}\ \mathsf{b}) \to \mathsf{t}\ \mathsf{a} \to \mathsf{m}\ (\mathsf{t}\ \mathsf{b})$$

Note that the type of algebras is unchanged!

Lookup with annotations

Same as before:

$$
\begin{aligned}
&\mathsf{lookup_{ALG}} :: \mathsf{Ord\ k} \Rightarrow \mathsf{k} \rightarrow \mathsf{Algebra\ (Tree_F\ k\ v)\ (Maybe\ v)} \\
&\mathsf{lookup_{ALG}\ k\ Leaf} = \mathsf{Nothing} \\
&\mathsf{lookup_{ALG}\ k\ (Branch\ n\ x\ l\ r)} = \textbf{case}\ \mathsf{k}\ \text{`}\mathsf{compare}\text{`}\ \mathsf{n}\ \textbf{of} \\
&\qquad \mathsf{LT} \rightarrow \mathsf{l} \\
&\qquad \mathsf{EQ} \rightarrow \mathsf{Just\ x} \\
&\qquad \mathsf{GT} \rightarrow \mathsf{r}
\end{aligned}
$$

Lookup now using cata_α:

$$
\begin{aligned}
&\mathsf{lookup}_\alpha :: (\mathsf{Ord\ k}, \mathsf{Out}\ \alpha\ (\mathsf{Tree_F\ k\ v})\ \mathsf{m}, \mathsf{Traversable}\ (\mathsf{Tree_F\ k\ v})) \Rightarrow \\
&\qquad \mathsf{k} \rightarrow \mu_\alpha\ \alpha\ (\mathsf{Tree_F\ k\ v}) \rightarrow \mathsf{m}\ (\mathsf{Maybe\ v}) \\
&\mathsf{lookup}_\alpha\ \mathsf{k} = \mathsf{cata}_\alpha\ (\mathsf{lookup_{ALG}\ k})
\end{aligned}
$$

Building trees

The function fromSortedList is an anamorphism:

$$\textbf{type}\ \mathsf{Coalgebra}\ \mathsf{f}\ \mathsf{s} = \mathsf{s} \to \mathsf{f}\ \mathsf{s}$$

$$\begin{aligned}\mathsf{ana}_\alpha &:: (\mathsf{In}\ \alpha\ \mathsf{f}\ \mathsf{m}, \mathsf{Monad}\ \mathsf{m}, \mathsf{Traversable}\ \mathsf{f}) \Rightarrow \\ &\quad\ \mathsf{Coalgebra}\ \mathsf{f}\ \mathsf{s} \to \mathsf{s} \to \mathsf{m}\ (\mu_\alpha\ \alpha\ \mathsf{f})\end{aligned}$$

$$\mathsf{ana}_\alpha\ \psi = \mathsf{in}_\alpha \lhd \mathsf{mapM}\ (\mathsf{ana}_\alpha\ \psi) \lhd \mathsf{return} \circ \psi$$

Building trees

The function fromSortedList is an anamorphism:

$$\textbf{type}\ \text{Coalgebra f s} = \text{s} \rightarrow \text{f s}$$

$$\text{ana}_\alpha :: (\text{In}\ \alpha\ \text{f m}, \text{Monad m}, \text{Traversable f}) \Rightarrow$$
$$\qquad \text{Coalgebra f s} \rightarrow \text{s} \rightarrow \text{m}\ (\mu_\alpha\ \alpha\ \text{f})$$
$$\text{ana}_\alpha\ \psi = \text{in}_\alpha \triangleleft \text{mapM}\ (\text{ana}_\alpha\ \psi) \triangleleft \text{return} \circ \psi$$

$$\text{fromSortedList} = \text{ana}_\alpha\ \text{fromSortedList}_{\text{ALG}}$$

Again, $\text{fromSortedList}_{\text{ALG}}$ is annotation-agnostic:

$$\text{fromSortedList}_{\text{ALG}} :: \text{Coalgebra}\ (\text{Tree}_{\text{F}}\ \text{k v})\ [(\text{k}, \text{v})]$$
$$\text{fromSortedList}_{\text{ALG}}\ [\,] = \text{Leaf}$$
$$\text{fromSortedList}_{\text{ALG}}\ \text{xs} =$$
$$\quad \textbf{let}\ (\text{l}, (\text{k}, \text{v}) : \text{r}) = \text{splitAt}\ (\text{length xs `div` } 2 - 1)\ \text{xs}$$
$$\quad \textbf{in}\ \text{Branch k v l r}$$

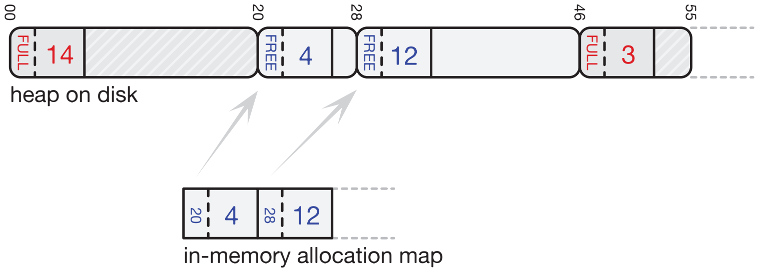
Heap

File-based storage heap

Linear list of blocks of binary data. Each block contains

- a used/free flag,
- a size,
- the payload as binary stream.

An in-memory allocation map is used for administration.



Heap interface

Most important operations:

$$\begin{aligned} \mathsf{read} &:: \mathsf{Binary}\ \mathsf{a} \Rightarrow \mathsf{Pointer}\ \mathsf{a} \rightarrow \mathsf{Heap}\ \mathsf{a} \\ \mathsf{write} &:: \mathsf{Binary}\ \mathsf{a} \Rightarrow \mathsf{a} \rightarrow \mathsf{Heap}\ (\mathsf{Pointer}\ \mathsf{a}) \end{aligned}$$

Running a heap computation:

$$\mathsf{run} :: \mathsf{FilePath} \rightarrow \mathsf{Heap}\ \mathsf{a} \rightarrow \mathsf{IO}\ \mathsf{a}$$

Persistence

Pointer annotation

$$\textbf{newtype}\ \mathsf{Ptr\ f\ a} = \mathsf{P}\ \{\mathsf{unP} :: \mathsf{Pointer\ (f\ a)}\}$$

The associated effect is accessing the heap:

$$\begin{aligned}
&\textbf{instance}\ (\mathsf{Binary}\ (\mathsf{f}\ (\mu_\alpha\ \mathsf{Ptr\ f}))) \Rightarrow \mathsf{Out\ Ptr\ f\ Heap}\ \textbf{where}\\
&\quad \mathsf{out}_\alpha = \mathsf{read} \lhd \mathsf{return} \circ \mathsf{unP} \circ \mathsf{out}\\
&\textbf{instance}\ (\mathsf{Binary}\ (\mathsf{f}\ (\mu_\alpha\ \mathsf{Ptr\ f}))) \Rightarrow \mathsf{In}\quad \mathsf{Ptr\ f\ Heap}\ \textbf{where}\\
&\quad \mathsf{in}_\alpha \quad = \mathsf{return} \circ \mathsf{In} \circ \mathsf{P} \lhd \mathsf{write}
\end{aligned}$$

Persistent operations

We specialize to the pointer annotation:

$$\textbf{type}\ \mathsf{Tree_P}\ \mathsf{k}\ \mathsf{v} = \mu_\alpha\ \mathsf{Ptr}\ (\mathsf{Tree_F}\ \mathsf{k}\ \mathsf{v})$$

$$\mathsf{fromSortedList_P} :: [(\mathsf{Int}, \mathsf{Int})] \to \mathsf{Heap}\ (\mathsf{Tree_P}\ \mathsf{Int}\ \mathsf{Int})$$
$$\mathsf{fromSortedList_P} = \mathsf{fromSortedList}$$

Persistent operations

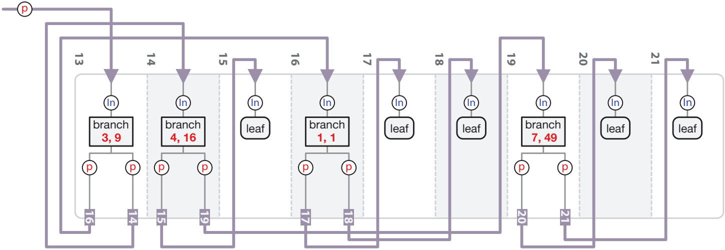
We specialize to the pointer annotation:

$$\textbf{type}\ \mathsf{Tree_P}\ \mathsf{k}\ \mathsf{v} = \mu_\alpha\ \mathsf{Ptr}\ (\mathsf{Tree_F}\ \mathsf{k}\ \mathsf{v})$$

$$\mathsf{fromSortedList_P} :: [(\mathsf{Int}, \mathsf{Int})] \to \mathsf{Heap}\ (\mathsf{Tree_P}\ \mathsf{Int}\ \mathsf{Int})$$
$$\mathsf{fromSortedList_P} = \mathsf{fromSortedList}$$

Example:

$$\mathsf{fromSortedList_P}\ [(1, 1), (3, 9), (4, 16), (7, 49)]$$



Using the system 1

BuildSquareDB.hs

```
main =
  do run "squares.db" $
       do p ← fromSortedList_P (map (λa → (a, a ∗ a)) [1 . . 10])
          storeRootPtr (p :: Tree_P Int Int)
     putStrLn "Database created."
```

$\mathsf{storeRootPtr} :: \mu_\alpha\ \mathsf{Ptr}\ \mathsf{f} \rightarrow \mathsf{Heap}\ ()$

Using the system 2

`LookupSquares.hs`

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

```
fetchRootPtr :: Heap (μα Ptr f)
```

Using the system 3

```
$ ghc --make BuildSquareDB.hs
$ ghc --make LookupSquares.hs
...
$ ./BuildSquareDB
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
...
$ ./LookupSquares
Give a number> 9
(9, Just 81)
Give a number> 12
(12, Nothing)
^C
$ _
```

More

In the paper and/or the thesis:

- Details about modification functions such as insert.
- How we deal with laziness and IO.
- How to extend the framework to higher-order fixed points (e.g., finger trees).

More

In the paper and/or the thesis:

- Details about modification functions such as insert.
- How we deal with laziness and IO.
- How to extend the framework to higher-order fixed points (e.g., finger trees).

Still to do:

- Sharing.
- Garbage collection.
- Concurrency.

Summary

Our framework allows you to:

- Define pure Haskell data structures.
- Generically annotate operations with effects.
- Save recursive data structures to the disk.

Unfortunately, you still have to:

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

The End

Modifying trees

The function insert is an apomorphism.

```
type ApoCoalgebra f s = s → f (Either s (μ f))

apo :: Functor f ⇒ ApoCoalgebra f s → s → μ f
apo ψ = In ∘ fmap apo′ ∘ ψ
  where apo′ (Left   l) = apo ψ l
        apo′ (Right r) = r
```

For every recursive position, we can decide if we want to continue with a new value, or if we want to place a tree.

Defining insert

The function insert modifies a given tree:

```
insert_ALG :: Ord k ⇒ k → v →
              ApoCoalgebra (Tree_F k v) (Tree k v)
insert_ALG k v (In Leaf) =
  Branch k v (Right (In Leaf)) (Right (In Leaf))
insert_ALG k v (In (Branch n x l r)) =
  case compare k n of
    LT → Branch n x (Left  l) (Right r)
    _  → Branch n x (Right l) (Left  r)
insert :: Ord k ⇒ k → v → Tree k v → Tree k v
insert k v = apo (insert_ALG k v)
```

We have to be more explicit about what parts of the old tree can be reused.

Partially annotated structures

$$\textbf{data}\ \text{Partial}\ \alpha\ \mathsf{f}\ \mathsf{a} = \text{New}\ (\mathsf{f}\ \mathsf{a})$$
$$\qquad\qquad\qquad\qquad \mid\ \text{Old}\ \ (\mu_{\alpha}\ \alpha\ \mathsf{f})$$

$$\textbf{type}\ \mu_{\widehat{\alpha}}\ \alpha\ \mathsf{f} = \mu_{\alpha}\ (\text{Partial}\ \alpha)\ \mathsf{f}$$

Endo-apomorphisms

```
type ApoCoalgebra f s = s → f (Either s (μ f))
apo :: Functor f ⇒ ApoCoalgebra f s → s → μ f
apo ψ = In ∘ fmap apo′ ∘ ψ
  where apo′ (Left  l) = apo ψ l
        apo′ (Right r) = r

type EndoApoCoalgebra_α α f =
  f (μ_α α f) → f (Either (μ_α α f) (μ_α̂ α f))

endoApo_α :: (OutIn α f m, Monad m, Traversable f) ⇒
             EndoApoCoalgebra_α α f → μ_α α f → m (μ_α α f)
endoApo_α ψ = outIn_α $ mapM endoApo_α′ ∘ ψ
  where endoApo_α′ (Left  l) = endoApo_α ψ l
        endoApo_α′ (Right r) = topIn r
topIn :: (In α f m, Monad m, Traversable f) ⇒
         μ_α̂ α f → m (μ_α α f)
```

Defining insert

No annotations:

```
insert_ALG :: Ord k ⇒ k → v → ApoCoalgebra (Tree_F k v) (Tree k v)
insert_ALG k v (In Leaf) =
  Branch k v (Right (In Leaf)) (Right (In Leaf))
insert_ALG k v (In (Branch n x l r)) =
  case compare k n of
    LT → Branch n x (Left  l) (Right r)
    _  → Branch n x (Right l) (Left  r)
```

With annotations:

```
insert_ALG :: Ord k ⇒ k → v → EndoApoCoalgebra_α α (Tree_F k v)
insert_ALG k v Leaf =
  Branch k v (make Leaf) (make Leaf)
insert_ALG k v (Branch n x l r) =
  case k `compare` n of
    LT → Branch n x (next l) (stop r)
    _  → Branch n x (stop l) (next r)
```