Generic Storage in Haskell

Sebastiaan Visser Andres Löh

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
sebastiaan@fvisser.nl andres@cs.uu.nl

Abstract

We present a framework for constructing functional data structures that can be stored on disk. The data structures reside in a heap saved in a binary file. Operations read and write only the parts of the data structure that are actually needed. The framework is based on expressing datatypes as fixed points of functors and then annotating the recursive positions with additional information. We explain how functions, if expressed in terms of standard recursion patterns, can be easily lifted from a pure setting to an effectful, annotated scenario. As a running example, we sketch how to implement a persistent library of finite maps based on binary search trees.

Categories and Subject Descriptors D.2.13 [*Reusable Software*]: Reusable libraries

General Terms Languages

Keywords datatype-generic programming, fixed points, annotations

1. Introduction

Algebraic datatypes in Haskell provide a powerful way to structure data. Recursive datatypes can be used to create functional data structures. Unfortunately, when data structures grow too large to fit in application memory or when the data outlives the running time of a single process there is no convenient way to store data structures outside application memory.

For most object-oriented programming languages there exist Object-Relational Mappers [2] that allow for a transparent mapping between objects and tables within relational databases. Automated derivation of database queries from the structure of objects can save time in the development process. Many attempts have been made to map values of algebraic datatypes in Haskell to relational databases tables. Due to the mismatch between the column based layout of relational databases and the structure of functional data structures only values of specific types can be marshalled.

In this paper we present a new framework for saving functional data structures in Haskell to a database file on disk. We do not use relational databases. Instead, we build our own system that relies on the structure of algebraic datatypes. We expose datatypes as fixed points of functors, and introduce the concept of effectful annotations. By writing operations as instances of specific recursion patterns such as catamorphisms and anamorphisms, we then obtain data structures that can be used in various ways. In particular, they can be used both in memory and on disk.

We identify the three important properties of our framework:

1. **Flexibility:** The storage system does not impose a single way to structure the data. Both general purpose and domain-specific data structures can be stored on disk.
2. **Efficiency:** By enabling incremental access to parts of the data we allow efficient manipulation of large collections of data. Algorithms working on a persistent data structure have the same asymptotic running time as their in-memory counterpart. Partial access is a signification extension to more conventional data serialization methods [15, 19].
3. **Transparency:** The final interface to the users uses common Haskell idioms. Users are not bothered with the implementation details of the storage system when manipulating persistent data structures.

This paper deals with the generic mapping from data structures that are used in memory to the same structures that can be stored on disk. The system we have implemented only works for single-threaded access to the data. Although we do not address the problem of concurrent access in detail, we quickly sketch what is needed to extend the framework to make multi threaded access possible.

Consider the following two simple Haskell programs that on a high level illustrate the use of our storage framework:

```
build :: IO ()
build = run "squares.db" $
  do let squares = [(1,1),(3,9),(4,16),(7,49)]
     produce (fromList_P squares)
find :: IO ()
find = run "squares.db" $ forever $
  do num ← liftIO (read `liftM` getLine)
     res  ← consume (lookup_P num)
     case res of
       Just sqr → liftIO (print (num, sqr))
       Nothing → modify (insert_P num (num ∗ num))
```

The first program opens a database file called `squares.db` and uses it to store a mapping from numbers to their squares. The second program – that can run independently from the first – opens up the same database and starts an interactive loop. In every iteration, the program expects a number and tries to look up the square of that number in the database. If a square is found, it will be reported back to the user; when no square is found, the square is computed and then added to the database.

The operations that are run against the database file run in their own monadic context, allowing to sequence multiple actions in one database action. We offer three basic operations to manipulate the database file: produce, consume and modify. The three functions are used to lift operations on persistent functional data structures to work on the database file. In the example program, we lift $\mathsf{fromList_P}$, $\mathsf{lookup_P}$ and $\mathsf{insert_P}$ to manipulate a persistent mapping from keys to values implemented as a binary search tree, similar to Haskell's `Data.Map` library [22].

In Section 2, we summarize the basic and well-known idea of expressing datatypes as fixed points of their pattern functors, and defining functions by instantiating recursion patterns such as catamorphisms and anamorphisms. We then show how to add annotations to the recursive positions of datatypes (Section 3) and how to associate functionality with the creation and removal of annotations. Instances of annotations have been used before, but we aim to describe (possibly effectful) annotations systematically here. In Section 4, we discuss how annotations affect recursion patterns and functions that are defined in terms of these patterns. We show that, in many cases, we can easily lift algebras written in a pure, annotation-agnostic style to work on annotated datatypes.

Our main contribution is the use of annotations for a generic storage framework. For that, we need an on-disk heap structure that can hold blocks of binary data (Section 5). It allows dynamic allocation and freeing of blocks on disk, and can grow and shrink on demand. We then use pointers as offsets to blocks on the storage heap as annotations for the recursive positions of datatypes (Section 6), yielding data structures than can be stored on disk. As a running example, we show how to apply the storage framework to create a persistent binary tree.

We discuss some subtleties of our approach as well as opportunities for future work in Section 7, present related work in Section 8 and conclude in Section 9.

2. Working with fixed points

In this section, we show how datatypes can be rewritten as fixed points, and algorithms working on such datatypes can be expressed in terms of recursion patterns [1, 16, 21, 24]. Reexpressing datatypes and algorithms in this style grants us fine-grained access to the structure of the datatypes, and thereby control over the behaviour of operations.

2.1 Recursive datatypes

As a running example of a typical recursive datatype, we consider the datatype of binary search trees:

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

The type Tree is parameterized over the type of keys k and the type of values v. The constructor Branch represents an internal node, containing a key, a value, a left and a right subtree. Leaves do not contain values and are represented by Leaf. We will maintain the binary search tree property as an invariant. For simplicity, we will not try to keep the tree properly balanced at all times.

An example tree, illustrated in Figure 1, can be defined as follows:

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

We now present some simple operations on binary search trees. As many functions that operate on datatypes, these examples follow the structure of the datatype closely: they are instances of standard recursion patterns.

First, let us consider the lookup function on binary search trees. Given a key, the function descends the tree. In each branch, the argument is compared to the stored key in order to decide what branch to take. If a correct key is found before a leaf is reached, the associated value is returned.

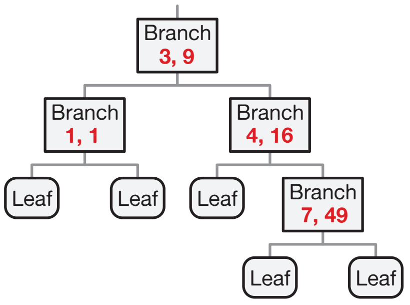

Figure 1. An example of a binary tree.

```
lookup :: Ord k ⇒ k → Tree k v → Maybe v
lookup _ Leaf               = Nothing
lookup k (Branch n x l r) = case k ‘compare‘ n of
                               LT → lookup k l
                               EQ → Just x
                               GT → lookup k r
```

Next, we define fromList, a function that creates a binary search tree from a list of key-value pairs. The function first sorts the list on the keys and then calls a helper function fromSortedList:

```
fromList :: Ord k ⇒ [(k, v)] → Tree k v
fromList = fromSortedList ∘ sortBy (comparing fst)
```

```
fromSortedList :: [(k, v)] → Tree k v
fromSortedList [ ] = Leaf
fromSortedList xs =
  let (l, (k, v) : r) = splitAt (length xs ‘div‘ 2 − 1) xs
  in Branch k v (fromSortedList l) (fromSortedList r)
```

If the input list is empty, a leaf is produced. Otherwise, we split the list into two parts of approximately equal length, use the middle pair for a new branch and call fromSortedList recursively for both the left and the right subtree.

Finally, we look at insert, a function that inserts a new key-value pair into a binary tree. Like lookup, the function performs a key comparison to ensure that the binary search tree property is preserved by the operation.

```
insert :: Ord k ⇒ k → v → Tree k v → Tree k v
insert k v Leaf             = Branch k v Leaf Leaf
insert k v (Branch n x l r) = case k ‘compare‘ n of
                                 LT → Branch n x (insert k v l) r
                                 _  → Branch n x l (insert k v r)
```

All three functions follow the structure of the Tree datatype closely. They recurse at exactly the places where the underlying datatype Tree is recursive. Function lookup destructs a tree, whereas fromList builds one. The function insert modifies a tree, or destructs one tree while building another.

2.2 Fixed points

We now show how abstracting from the recursive positions in a datatype and re-expressing the datatype as a fixed point of a functor helps us to make the recursion patterns of the operations explicit. For our running example, this means that we move from Tree to $\mathsf{Tree_F}$ by adding a parameter r that is used wherever Tree makes a recursive call:

```
data Tree_F k v r = Leaf | Branch k v r r
```

The type $\mathsf{Tree_F}$ is also called the *pattern functor* of Tree.

To get our binary search trees back, we have to tie the recursive knot, i.e., instantiate the parameter r with the recursive call. This job is performed by the type-level fixed point combinator μ that takes a functor f of kind $* \to *$ and parameterizes f with its own fixed point:

```
newtype μ f = In {out :: f (μ f)}
```

By using μ on a pattern functor such as $\mathsf{Tree_F}$, we obtain a recursive datatype once more that is isomorphic to the original Tree datatype:

```
type Tree k v = μ (Tree_F k v)
```

Building a binary tree structure for our new Tree type requires wrapping all constructor applications with an additional application of the In constructor of the μ datatype. It is thus helpful to define "smart constructors" for this task:

```
leaf :: Tree k v
leaf = In Leaf

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

Our example tree can now be expressed in terms of leaf and branch rather than Leaf and Branch, but otherwise looks as before.

2.3 Recursion patterns

Given a fixed point representation of a datatype, we can define a number of recursion patterns for the datatype. But first, we have to back up the fact that the pattern functor really is a functor. To this end, we make it an instance of the Functor class:[1]

```
instance Functor (Tree_F k v) where
  fmap _ Leaf              = Leaf
  fmap f (Branch k v l r) = Branch k v (f l) (f r)
```

Catamorphism A *catamorphism* is a recursion pattern that consumes a value of a given data structure systematically. It is a generalization of Haskell's foldr function to other datatypes:

```
type Algebra f r = f r → r

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

The argument to the catamorphism is often called an *algebra*. The algebra describes how to map a functor where the recursive positions have already been evaluated to a result. The function cata then repeatedly applies the algebra in a bottom-up fashion to transform a whole recursive structure.

The function lookup on binary search trees is an example of a catamorphism. An algebra for lookup is defined as follows:

```
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
```

Compared to the original definition of lookup, the definition of $\mathsf{lookup_{ALG}}$ is not recursive. The benefit for us is that the new form facilitates changing the behaviour of the function at recursive calls.

We can get the behaviour of the original lookup function back by running the algebra – we simply pass it to the cata function:

```
lookup :: Ord k ⇒ k → Tree k v → Maybe v
lookup k = cata (lookup_ALG k)
```

Anamorphism An *anamorphism* is a recursive pattern that is dual to the catamorphism. Where the catamorphism systematically decomposes a structure, the anamorphism systematically builds one. It is a generalization of Haskell's unfoldr function to other datatypes:

```
type Coalgebra f s = s → f s

ana :: Functor f ⇒ Coalgebra f s → s → μ f
ana ψ = In ∘ fmap (ana ψ) ∘ ψ
```

The argument to an anamorphism is called a *coalgebra*. A coalgebra takes a seed of type s and produces a functor f s where the elements contain new seeds. The function ana repeatedly runs the coalgebra to all the seed values, starting from the original seed, until none remain.

The function fromSortedList is an anamorphism on trees. We can define a suitable coalgebra as follows:

```
fromSortedList_ALG :: Coalgebra (Tree_F k v) [(k, v)]
fromSortedList_ALG [] = Leaf
fromSortedList_ALG xs =
  let (l, (k, v) : r) = splitAt (length xs `div` 2 − 1) xs
  in Branch k v l r
```

The definition is very similar to the original one, but again, we have no recursive calls, as those are handled by ana now:

```
fromList :: Ord k ⇒ [(k, v)] → Tree k v
fromList = ana fromSortedList_ALG ∘ sortBy (comparing fst)
```

Apomorphism Let us recall the function insert. When we are in a branch of the tree, we compare the stored key with the given key. Depending on the outcome, we continue inserting into one subtree, but want to keep the other subtree unchanged. While it is possible to coerce insert into both the catamorphism and the anamorphism pattern, neither pattern is a very good fit.

Instead, we use an *apomorphism* [30] – a generalization of an anamorphism, and the dual concept of a paramorphism [20].

```
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
```

Where the coalgebra for anamorphisms generates a functor with new seeds from a single seed, we can now decide at every recursive position whether we want to produce a new seed (Left), or whether we simply want to provide a recursive structure to use at this point (Right).

It is now easy to define a coalgebra for insert:

```
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 now introduced three useful patterns that allow us to define functions on a fixed point representation of a datatype in such a way that we abstract from the recursive structure. In the next section, we will make use of the abstraction by adding new functionality in the form of annotations to the recursive positions.

3. Annotations

By moving from a recursive datatype to its pattern functor, we now have control over what exactly to do with recursive positions. We can simply tie the knot using the fixed point combinator μ, as we

[1] Since version 6.12.1, GHC can derive this instance automatically.

have seen above. However, we can also store additional information at each recursive position. In this section, we discuss how we can move from fixed poins to annotated fixed points. We discuss how to create and remove annotations systematically, and discuss example annotations.

3.1 Annotated fixed points

The *annotated* fixed point combinator μ_α is defined as follows:

```
type μ_α α f = μ (α f)
```

Instead of taking the fixed point of f, we take the fixed point of α f, where α is a type constructor (of kind $(* \to *) \to * \to *$) that can be used to modify the functor, for example by adding additional information.

The simplest annotation is the *identity annotation*:

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

Using μ_α Id is isomorphic to using μ.

We can define an annotated variant of binary search trees by applying μ_α in place of μ:

```
type Tree_α α k v = μ_α α (Tree_F k v)
```

Once again, Tree_α Id is isomorphic to our old Tree type.

3.2 Creating and removing annotations

Our main goal in this article is to use annotations to represent pointers to data that is stored on disk. Reading and writing to disk are effectful operations. Therefore, we allow the creation and removal of annotations to be associated with a monadic context.

We now define type classes In and Out that generalize the In and out operations on fixed points to the annotated scenario. The method in_α wraps a functor with fully annotated substructures and adds a new annotation. The out_α method unwraps an annotated node, exposing the functor with the annotated substructures. The results of both operations live in a monad m:

```
class Monad m ⇒ In α f m where
  in_α :: f (μ_α α f) → m (μ_α α f)
class Monad m ⇒ Out α f m where
  out_α :: μ_α α f → m (f (μ_α α f))
```

The functor f is added as a class parameter, so that we can impose additional restrictions on it at a later stage.

For the identity annotation, we choose m to be the identity monad (called Identity), and in_α and out_α are in essence just In and out:

```
instance In Id f Identity where
  in_α = return ∘ In ∘ Id
instance Out Id f Identity where
  out_α = return ∘ unId ∘ out
```

Using the In type class, we define two new smart constructors for the annotated binary search tree datatype:

```
leaf_α :: In α (Tree_F k v) m ⇒ m (Tree_α α k v)
leaf_α = in_α Leaf
branch_α :: In α (Tree_F k v) m ⇒
            k → v → Tree_α α k v → Tree_α α k v →
            m (Tree_α α k v)
branch_α k v l r = in_α (Branch k v l r)
```

The leaf_α and branch_α smart constructors can be used to build up annotated binary search trees for an arbitrary annotation type α. However, since the annotation type is associated with a monadic context, we now have to build our example tree in monadic style:

```
myTree_α :: In α (Tree_F Int Int) m ⇒ m (Tree_α α Int Int)
myTree_α =
  do l ← leaf_α
     d ← branch_α 7 49 l  l
     e ← branch_α 1 1   l  l
     f ← branch_α 4 16 d l
     branch_α 3 9 e f
```

Note the type of myTree_α: the value is overloaded on the annotation α, so we can use it with different annotations later.

3.3 Example annotation: modification time

As a non-trivial example of an annotation, let us keep track of the modification time of substructures. For this purpose, we define a new datatype ModTime that can be used as an annotation: next to the actual structure, it also saves a LocalTime.[2]

```
data ModTime f a = M {time :: LocalTime, unM :: f a}
```

In order to use the annotation, we have to define instances of both the In and Out classes, and thereby specify the behaviour associated with creating and removing the annotation. In our case, we want to store the current time when creating the annotation, but do nothing further when dropping it:

```
instance In ModTime f IO where
  in_α f = do t ← getCurrentTime
              return (In (M t f))
instance Out ModTime f IO where
  out_α = return ∘ unM ∘ out
```

Because getting the current time requires a side effect, the ModTime annotation is associated with the IO monad.

We can now use the annotation, for example with our binary search tree type, which we specialize to use the ModTime annotation:

```
type TreeM k v = Tree_α ModTime k v
```

As a simple use case, we can evaluate the overloaded example tree myTree_α using the modification time annotation simply by specializing its type accordingly. The construction of the tree has to take place in the IO monad:

```
ghci> myTree_M <- myTree_a :: IO (TreeM Int Int)
{M 236807 (Branch 3 9
  {M 236755 (Branch 1 1
    {M 236688 Leaf}
    {M 236688 Leaf})}
  {M 236781 (Branch 4 16
    {M 236728 (Branch 7 49
      {M 236688 Leaf}
      {M 236688 Leaf})}
    {M 236688 Leaf})})}
```

For readability, we have cropped the modification times to the microseconds, reformatted the output slightly to resemble the tree structure, and used a custom Show instance for μ that uses curly braces for the In constructor. The tree is shown schematically in Figure 2. We bind the result of the computation to `myTree_D` for later reuse.

The last modification times of all the leaves are the same, because we share the result of one call to leaf_α in the definition of myTree_α. We see that the modification times follow the order of the monadic operations in myTree_α: the leaves are created first, the root of the tree is created last.

3.4 Example annotation: debug trace

As another example of an annotation we introduce Debug:

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

This annotation does not store any data except for the functor itself. We are only interested in the effects associated with creation and

[2] The LocalTime type is from the Haskell `time` package.

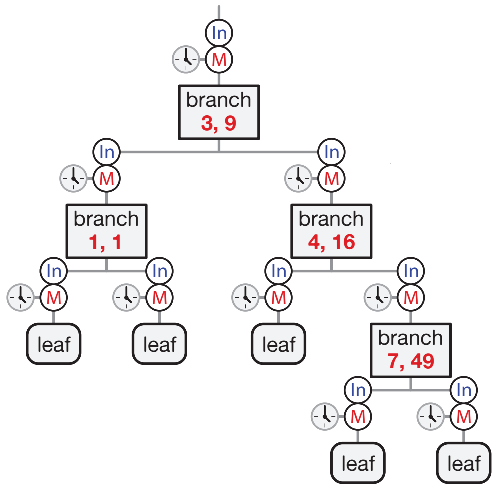

Figure 2. Binary tree with local modification times saved as annotations at the recursive positions.

removal of the annotation: we want to create a debug trace of these operations when they occur.

The desired behaviour is implemented in the In and Out instances for the debug annotation:

instance (Functor f, Show (f ())) ⇒ In Debug f IO **where**
  in_α = return ∘ In ∘ D ◁ printer "in"

instance (Functor f, Show (f ())) ⇒ Out Debug f IO **where**
  out_α = printer "out" ∘ unD ∘ out

Here,

(◁) :: Monad m ⇒ (b → m c) → (a → m b) → a → m c
(f ◁ g) x = g x ≫= f

is right-to-left Kleisli composition.[3] It has lower precedence than normal function composition. The function printer prints the top layer of a given recursive structure f and also returns f:

printer :: (Functor f, Show (f ())) ⇒ String → f a → IO (f a)
printer s f = print (s, fmap (const ()) f) ≫ return f

To use the debug annotation, we first specialize our binary tree type:

type TreeD k v = Tree_α Debug k v

Then, we evaluate myTree_α once more, this time using TreeD as a result type. This causes a trace of all the contruction steps to be printed:

```
ghci> myTree_D <- myTree_a :: IO (TreeD Int Int)
("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}
{D Leaf})} {D (Branch 4 16 {D (Branch 7
49 {D Leaf} {D Leaf})} {D Leaf})})}
```

3.5 Summary

In this section, we have introduced annotated fixed points. We have discussed how to abstract from the creation and removal of annotations by means of the In and Out type classes. We have also introduced two example annotations, both with effects in the IO monad: one to keep track of the modification time of substructures, and one to generate a debug trace of operations on the structure. In Section 6, we will introduce an annotation that allows us to make data structures persistent.

Before that, we have a closer look at how to work with annotated structures. Until now, we have seen that values defined in a monadic style such as myTree_α can be evaluated at different annotation types, leading to different behaviour. In the next section, we show how the recursion patterns we introduced in Section 2.3 can be lifted to the annotated scenario, meaning that we can also lift functions specified via algebras to work on annotated structures easily.

4. Annotated recursion patterns

It is inconvenient to write operations on annotated datatype directly. Since both in_α and out_α are monadic, we are forced to use monadic style everywhere. Furthermore, code that is supposed to be generic in the annotation type cannot use pattern matching, because nothing is known about the shape of the annotation.

In this section, we show that by using recursion patterns, we can avoid the above problems. We demonstrate that catamorphisms, anamorphisms and apomorphisms can all be easily adapted to work with annotated structures. As we will see, in many cases we can reuse the original algebras, written in a pure, annotation-agnostic way. By plugging these algebras into the new patterns, they can run in a setting where effectful operations are performed behind the scenes.

4.1 Catamorphism

Recall the definition of a catamorphism from Section 2.3:

type Algebra f r = f r → r

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

In order to make the function annotation-ready, we replace out by out_α; as a consequence, everything becomes monadic [7], so we replace function composition by Kleisli composition; finally, we replace fmap by mapM:

cata_α :: (Out α f m, Monad m, Traversable f) ⇒
    Algebra f r → μ_α α f → m r
cata_α ϕ = return ∘ ϕ ◁ mapM (cata_α ϕ) ◁ out_α

Haskell's Traversable type class replaces the Functor constraint – it contains the mapM method. Note that we use the same Algebra type as before, and assume pure algebras that are defined in an annotation-agnostic way – exactly as we want.

Before we can use cata_α on an actual datatype such as binary search trees, we have to give a Traversable instance for the pattern functor:[4]

instance Traversable (Tree_F k v) **where**
  mapM _ Leaf   = return Leaf
  mapM f (Branch k v l r) = liftM2 (Branch k v) (f l) (f r)

As before, we obtain an actual lookup function by passing the algebra to cata_α:

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

We can use lookup once we have an annotated tree. We reuse `myTree_M` that is bound to the result of evaluating myTree_α using the modification time annotation. The following expression returns the expected result, but now in the IO monad:

[3] Available as <=< in Haskell.

[4] Haskell's Traversable has Foldable as superclass, so we have to define that instance as well, but since its functionality is not used here, we omit it. GHC 6.12.1 and later can derive both Foldable and Traversable automatically.

```
ghci> lookup 4 myTree_M
Just 16
```

However, if we use `myTree_D` that is bound to the result of evaluating myTree_α in the debug annotation, the call to lookup reveals a problem:

```
ghci> lookup 4 myTree_D
("out",Branch 3 9 () ())
("out",Branch 1 1 () ())
("out",Leaf)
("out",Leaf)
("out",Branch 4 16 () ())
("out",Branch 7 49 () ())
("out",Leaf)
("out",Leaf)
("out",Leaf)
Just 16
```

The function produces a result and a trace as expected. However, the trace reveals that the *entire* tree is traversed, not just the path to the Branch containing the key 4. The culprit is the strictness of IO, that propagates to the whole operation now that IO is used behind the scenes. We defer the discussion of this problem until Section 7.1.

We can use the annotated catamorphism also to remove all annotations from a recursive structure, removing all layers of annotations and performing the associated effects:

```
fullyOut_α :: (Out α f m, Monad m, Traversable f) ⇒
          μ_α α f → m (μ f)
fullyOut_α = cata_α In
```

4.2 Anamorphism

For anamorphisms, the situation is very similar as for catamorphisms. We define an annotated variant of ana, called ana_α, by lifting everything systematically to the annotated monadic setting:

```
type Coalgebra f s = s → f s
ana_α :: (In α f m, Monad m, Traversable f) ⇒
        Coalgebra f s → s → m (μ_α α f)
ana_α ψ = in_α ◁ mapM (ana_α ψ) ◁ return ∘ ψ
```

Note that the Coalgebra type synonym is unchanged and just repeated here for convenience.

We can now produce annotated values easily. Instead of using a monadic construction such as in the definition of myTree_α, we can resort to fromList:

```
fromList xs = ana_α fromSortedList_ALG
                    (sortBy (comparing fst) xs)
```

The definition

```
myTree_α′ :: In α (Tree_F Int Int) m ⇒ m (Tree_α α Int Int)
myTree_α′ = fromList [(1,1),(3,9),(4,16),(7,49)]
```

is equivalent to the old myTree_α, but significantly more concise.

The counterpart to $\mathsf{fullyOut}_\alpha$ that completely removes all annotations from a structure is $\mathsf{fullyIn}_\alpha$ that completely annotates a recursive structure:

```
fullyIn_α :: (In a f m, Monad m, Traversable f) ⇒
         μ f → m (μ_α a f)
fullyIn_α = ana_α out
```

4.3 Apomorphism

For the apomorphism, we have slightly more work to do, because the fixed point combinator occurs in the type of coalgebras:

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

As a first step, we are changing the coalgebra type to use μ_α instead:

```
type ApoCoalgebra_α α f s = s → f (Either s (μ_α α f))
```

We can now define apo_α:

```
apo_α ::  (In α f m, Monad m, Traversable f) ⇒
          ApoCoalgebra_α α f s → s → m (μ_α α f)
apo_α ψ = in_α ◁ mapM apo_α′ ◁ return ∘ ψ
  where apo_α′ (Left  l) = apo_α ψ l
        apo_α′ (Right r) = return r
```

Unfortunately, we cannot directly use apo_α to lift insert to work on annotated binary search trees. The reason is that a modification function such as insert both destructs and constructs a tree. If we want to use an annotated binary search tree as seed for the apomorphism, we have to destruct it in the coalgebra in order to pattern match – but we cannot, because destructing is associated with effects. Furthermore, we create new leaves when inserting key-value pairs – but again, we cannot, because constructing new annotated values is associated with effects.

We therefore define a new recursion pattern for modifiers such as insert in Section 4.5. As a preparation for the new pattern, we first look at partially annotated structures.

4.4 Partially annotated structures

Let us capture the idea of building some layers of a recursive structure in a pure, annotation-agnostic way, while still being able to reuse parts of an annotated structure that we have available already.

To this end, we introduce an *annotation transformer* called Partial. Given an annotation α, we can either choose to create a new unannotated layer, or (re)use a complete annotated subtree:

```
data Partial α f a = New (f a)
                   | Old  (μ_α α f)
```

We define an abbreviation for partially annotated structures:

```
type μ_α̂ α f = μ_α (Partial α) f
```

Using the function topIn, we can complete the missing annotations at the top of a partially annotated structure:

```
topIn :: (In α f m, Monad m, Traversable f) ⇒
     μ_α̂ α f → m (μ_α α f)
topIn = topIn′ ∘ out
  where topIn′ (New x) = (in_α ◁ mapM topIn) x
        topIn′ (Old  x) = return x
```

4.5 Modification functions

By employing the partially annotated structures, we can now introduce a variant of the apomorphism that modifies a given structure.

As a preparation, we define a type class OutIn that combines the functionality of the Out and In classes: using the class method outIn_α, an annotated node is unwrapped, modified, and finally re-wrapped:

```
class (Out α f m, In α f m) ⇒ OutIn α f m where
  outIn_α  ::  (f (μ_α α f) → m (f (μ_α α f)))
           →   (μ_α α f) → m    (μ_α α f)
  outIn_α f = in_α ◁ f ◁ out_α
```

We require both Out and In as superclasses of OutIn. Given the out_α and in_α methods, we supply a default implementation for outIn_α. For all the annotations we have been using so far, the default implementation is sufficient:

```
instance OutIn Id f Identity
instance OutIn ModTime f IO
instance (Functor f, Show (f ())) ⇒ OutIn Debug f IO
```

For some annotation types (such as the Heap annotation we use in Section 6), we can give an improved direct definition.

We can now define a variant of the apomorphisms for modification functions that is different from the normal apomorphism in the following ways:

- the type of the seed is restricted to be a value of the recursive structure itself – this also motivates the name *endo-apomorphism* for the new pattern;
- instead of ending the recursion by returning a fully annotated tree, we allow to stop with a partially annotated tree.

We give both the old and the new coalgebra type for comparison:

```
type ApoCoalgebra_α        α f s =
   s            → f (Either s          (μ_α α f))
type EndoApoCoalgebra_α α f   =
   f (μ_α α f) → f (Either (μ_α α f) (μ_α̂ α f))
```

The associated recursion pattern looks as follows:

```
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
```

Compared to the regular apomorphism, we use outIn$_\alpha$ because we now work with the same source and target structure, and we use topIn to create the missing annotations once we stop the recursion.

When defining a coalgebra for use with endoApo$_\alpha$, we now effectively have the choice between the following three actions per recursive position:

- We can produce a new seed to drive the next recursive step, by selecting the left part of the sum type in the result of the coalgebra. We define a helper function with the more meaningful name next for this choice:

  ```
  next :: μ_α α f → Either (μ_α α f) (μ_α̂ α f)
  next = Left
  ```

- We can reuse a fully annotated part of the input as output, stopping the recursion at this point. We define the helper function stop for this purpose:

  ```
  stop :: μ_α α f → Either (μ_α α f) (μ_α̂ α f)
  stop = Right ∘ In ∘ Old
  ```

- Finally, we can create one or more layers of new nodes, by using the helper function make, which takes a partially annotated structure as its argument:

  ```
  make :: f (μ_α̂ α f) → Either (μ_α α f) (μ_α̂ α f)
  make = Right ∘ In ∘ New
  ```

Let us now return to our example, the insert function on binary search trees. Unfortunately, we cannot quite reuse the original coalgebra we have given in Section 2.3. We have to be more explicit about where we reuse old parts the tree, and where we create new parts of the tree:

```
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)
```

The differences are relatively minor: We can still define insert$_{\mathrm{ALG}}$ as a pure function, and annotation-agnostic. We no longer have to pattern match on parts of the recursive structure, because we use the endo-apomorphism now. And we can still get the original behaviour back, by specializing to the identity annotation and the identity monad.

We run insert$_{\mathrm{ALG}}$ by passing it to endoApo$_\alpha$:

```
insert k v = endoApo_α (insert_ALG k v)
```

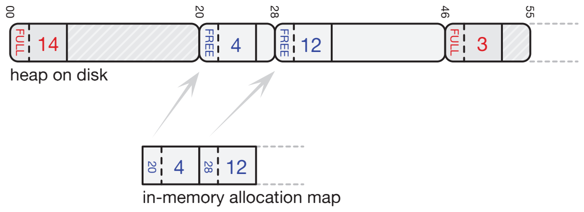

Figure 3. A snippet of a heap structure containing four blocks of which two block are in use and contain a payload. The blocks are placed next to each other. An in-memory allocation map is used to map payload sizes to free blocks of data.

4.6 Summary

In this and the previous section we have shown a framework for generically annotating recursive datatypes. Using an annotated fixed point combinator we are able to store custom markers (containing potentially custom information) at the recursive positions of functional data structures. We associate extra functionality – potentially with effects – with the creation, removal and modification of annotated recursive structures.

By defining algebras for specific recursion patterns, we can define functions in a pure style, without having to worry about annotations or monadic contexts. We have shown a number of frequently occurring patterns, for consumers, for producers, and for modifiers. More patterns can be defined in a similar style if desired. For example, we can define an endo-paramorphism that is dual to the endo-apomorphism.

We have shown that several operations on binary search trees can be expressed using such patterns. In fact, we have built a library that replicates most of a finite map data structure based on binary search trees in our annotated framework, yielding a data structure that can be flexibly used with several annotations.

5. File-based storage heap

In the previous sections, we showed how to perform generic programming with fixed point annotations. The annotations form the basis of our storage framework. We use the annotations to marshal individual nodes from and to a database file on disk. Before we can explain this storage annotation in more detail, we first sketch the low-level storage layer.

In this section, we introduce a block-based heap data structure that is used to allocate and use fragments of binary data on disk. The structure of the heap is similar to that of in-memory heaps as used by most programming languages to manage dynamically allocated data. The heap structure can freely grow and shrink on demand.

The heap uses a file to store a contiguous list of blocks of binary data. Each of the blocks contains a header and a payload. The header contains a flag to tell if the block is currently free or in use. Furthermore, the header indicates the size of the block. The payload is an arbitrary sequence of binary data. The size of the payload must not exceed the size specified in the header minus the header size. An example layout of the heap is shown in Figure 3.

The heap described here has a rather imperative and low-level implementation. Some of the operations are unsafe: they have invariants that are not enforced at compile time. We emphasize that these unsafe operations are internal to our framework. The user defines operations in terms of a safe and more high-level interface as discussed in the next section.

5.1 Offset pointers

Applications that use the heap can allocate blocks of data of any size and use it for writing and reading data. All access to the heap is managed using pointers. The pointer datatype just stores an integer that represents an offset into the heap file. An invariant is that pointers always point to the beginning of a block.

```
type Offset = Integer

newtype Ptr (f :: * → *) a = P Offset
  deriving Binary
```

The Ptr has two phantom type arguments f and a that are used to ensure that only values of type f a can be written to or read from the location addressed by the pointer. By using two type arguments rather than one, we ensure that Ptr has the right kind to be used as a fixed-point annotation.

5.2 Heap context

All heap operations run inside a monadic Heap context. The Heap monad is a monad transformer stack that uses the IO monad on the inside:

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

The context uses a reader monad to distribute the file handle of the heap file to all operations, and it makes use of a state monad to manage an *allocation map*.

The allocation map stores a mapping of block sizes to the offsets of all blocks that are not currently in use. Because we manage an in-memory map, no disk access is needed when allocating new blocks of data.

From the point of view of the user the Heap is opaque, no access to the internals of the monad are required to work with the heap. In order to run a sequence of heap operations we use the run function, which receives the name of a heap file:

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

Because of the file access, the result of run is in the IO monad. The run function opens the heap file and initializes it if it is new. If the file exists, it quickly scans all blocks to compute the in-memory allocation map. It then applies the heap computations, and closes the heap file in the end.

5.3 Heap operations

We now present the interface of the heap structure. For each operation, we provide the type signature and a short operations, but we do not go into details about the implementation.

- allocate :: Integer → Heap (Ptr f a)

  The allocate operation can be used to allocate a new block of data that is large enough to hold a payload of the given size. The function can be compared to the in-memory malloc function from the C language. On return, allocate yields a pointer to a suitable block on disk. The function marks the current block as occupied in the in-memory allocation map. Subsequent calls to allocate will no longer see the block as eligible for allocation.

- free :: Ptr f a → Heap ()

  When a block is no longer needed, it can be freed using the free operation. The internal allocation map will be updated so the block can be reused in later allocations.

- write :: Binary (f a) ⇒ f a → Heap (Ptr f a)

  The write operation takes a Haskell value, serializes it to a binary stream, allocates just the right amount of data on the heap and then stores the value in the block on disk.

  To produce a binary serialization of a Haskell value, the Binary type class is used [19]. The interface of the class is as follows:

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

  The put method serializes a value to a binary stream, whereas get deserializes a binary stream back to a Haskell value.[5]

- read :: Binary (f a) ⇒ Ptr f a → Heap (f a)

  Dual to the write operation, we have the read operation. The function takes a pointer to a block on disk, reads the binary payload, deserializes the payload to a Haskell value using the Binary type class and returns the value.

- fetch :: Binary (f a) ⇒ Ptr f a → Heap (f a)

  The fetch operation is a variant of the read operation. Whereas read leaves the original block intact, fetch frees the block after reading the data.

- writeRoot :: Ptr f a → Heap ()

  The writeRoot operation is a variant of write that will show to be useful in the next section. The function takes a pointer to some block on the heap and will store *the pointer value* on a fixed location on the heap. This function can be used to store the root of a data structure in a place that can easily be identified, also beyond the end of a database session.

- readRoot :: Heap (Ptr f a)

  The readRoot operation can be used to read back the pointer that has been stored by the writeRoot operation.

The allocate and free heap operations are both used in the implementation of write, read, fetch, but are not used further in this framework. In the next section we see how the five higher-level functions write, read, fetch, writeRoot and readRoot can be used in combination with the annotation framework to build persistent data structures.

5.4 Summary

In this section, we have sketched the interface of a file-based heap structure. It can be used to store arbitrary blocks of binary data on disk. Access to the data is managed by pointers as offsets into the file. All Haskell values that have an instance for the Binary type class can automatically be marshalled from and to the heap. The heap structure is low-level and does not assume anything about the contents of the individual blocks.

6. Persistent data structures

Everything is in place now to define an annotation that allows us to make data structures persistent. In the previous section, we have chosen to define the Ptr datatype with *two* type parameters – a functor of kind $* \to *$ and an explicit index of kind $*$. This design decision makes Ptr usable as a fixed point annotation. Creating a Ptr corresponds to writing to the heap, whereas removing a Ptr implies reading from the heap.

We can then build concrete persistent data structures such as binary search trees, by using the pointer annotation:

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

When we work with a value of type $\mathsf{Tree_P}\ k\ v$, we now actually work with a *pointer* to a binary tree that lives somewhere on the heap that is stored on the disk. To be precise, the pointer references a heap block that stores a binary serialization of a single node of type $\mathsf{Tree_F}\ k\ v\ (\mathsf{Tree_P}\ k\ v)$. The recursive positions of the node contain

[5] Both Get and Put are monads defined in the Binary class. The details are not relevant for our purposes here. Using the `regular` [28] library for generic programming, we have created a generic function that can be used to automatically derive Binary instances for Haskell datatypes that are regular.

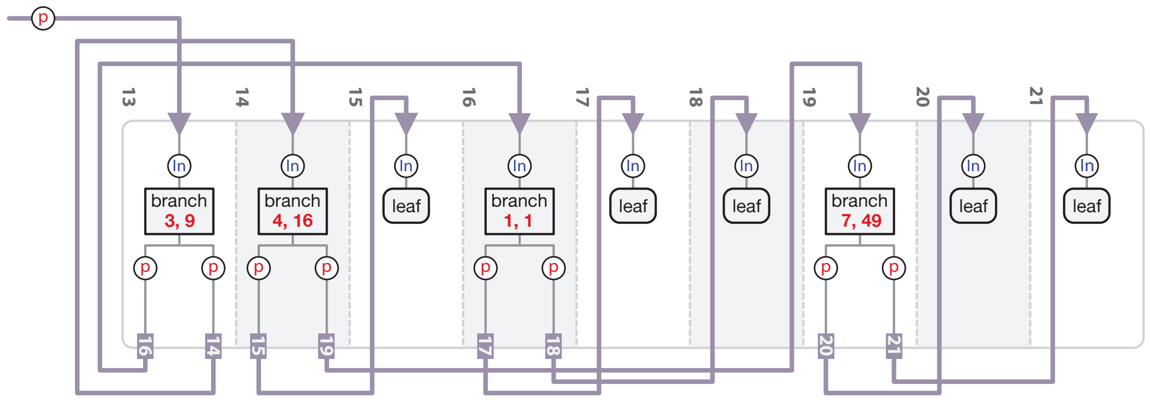

Figure 4. A persistent binary tree that lives on the storage heap. Each node is stored on its own heap block in binary representation. All substructures are referenced by pointer to the file offset.

again pointers to substructures. Figure 4 shows how such a tree looks like.

6.1 Persistent producers and consumers

To make the pointer type Ptr usable as an annotation, we have to define instances of the Out and In type classes from Section 3. We associate the pointer annotation with the Heap context, use the read operation as the implementation for out_α and use the write operation as the implementation for in_α:

```
instance (Traversable f, Binary (f (μ_α Ptr f))) ⇒
        Out Ptr f Heap
  where out_α = read ◁ return ∘ out
instance (Traversable f, Binary (f (μ_α Ptr f))) ⇒
        In Ptr f Heap
  where in_α = return ∘ In ◁ write
```

To make the two instances work, we need a Binary instance for both the fixed point combinator and the $\mathsf{Tree_F}$ pattern functor:

```
instance Binary (f (μ f)) ⇒ Binary (μ f) where
  put (In f) = put f
  get = fmap In get
instance (Binary k, Binary v, Binary f) ⇒
        Binary (Tree_F k v f) where
  put Leaf             = do putWord8 0
  put (Branch k v l r) = do putWord8 1
                           put k; put v; put l; put r
  get = do t ← get
          if t ≡ (0 :: Word8)
            then return Leaf
            else liftM4 Branch get get get get
```

We can now specialize the fromList function from Section 4.3 to use the pointer annotation in the Heap context. This yields an operation that builds a binary search tree *on disk* instead of in application memory:

```
fromList_P :: [(Int, Int)] → Heap (Tree_P Int Int)
fromList_P = fromList
```

Note that all we have to do is to specialize the type of the original operation. We can reuse exactly the same fromList function.

The result of running the fromList operation against a heap file is a pointer to the root node of the tree as stored on disk, wrapped inside an In constructor. Performing the operation is as simple as supplying it to the run function from our heap interface:

```
ghci> let squares = [(1,1),(3,9),(4,16),(7,49)]
ghci> run "squares.db" (fromListP squares)
```

Figure 4 shows an illustration of our example tree laid out on the heap. The example above writes a binary tree of integers to disk as expected, but has a slight problem when used on its own: the root pointer of the structure is discarded and lost. We therefore define a helper function produce that takes a producer operation, such as $\mathsf{fromList_P}$, runs the operation on the heap and then saves the final pointer in a reserved location on the heap:

```
produce :: Binary (f (μ f)) ⇒ Heap (μ f) → Heap ()
produce c = c ≫= writeRoot ∘ out
```

We write the pointer to the root node of the produced data structure to a special location on disk, we call this location the *root block*. Becaues we use writeRoot to store the root pointer we can easily read back the pointer using readRoot to perform consecutive operations on the same data structure. We delete the `squares.db` file and run the example again, this time saving the root node:

```
ghci> run "squares.db" (produce (fromListP squares))
```

We make a similar wrapper function for performing consumer functions. The consume operation takes a heap operation and passes it as input the data structure pointed to by the pointer stored in the root block:

```
consume :: Binary (f (μ f)) ⇒ (μ f → Heap b) → Heap b
consume c = readRoot ≫= c ∘ In
```

We can now run any consumer operation on the binary tree of squares stored on disk. We specialize the lookup function and apply it to our squares database:

```
lookup_P :: Int → Tree_P Int Int → Heap (Maybe Int)
lookup_P = lookup
```

```
ghci> run "squares.db" (consume (lookupP 3))
Just 9
```

The database file is opened and the root pointer is read from the root block. The root pointer references a persistent binary tree that is passed to the $\mathsf{lookup_P}$ function that, node by node, traverses the tree until the key is found and the value can be returned.

6.2 Persistent modification

We have described how producers such as fromList and consumers such as lookup can easily be lifted to a persistent setting if defined in our generic annotation framework. We now show the same for modifiers.

To start, we have to give an instance for the OutIn type class for the pointer annotation in the Heap context:

$$\begin{aligned}&\textbf{instance}\ (\mathsf{Traversable\ f}, \mathsf{Binary}\ (\mathsf{f}\ (\mu_\alpha\ \mathsf{Ptr\ f}))) \Rightarrow\\&\qquad \mathsf{OutIn\ Ptr\ f\ Heap}\\&\quad \textbf{where}\ \mathsf{outIn}_\alpha\ \mathsf{f} = \mathsf{return} \circ \mathsf{In} \lhd \mathsf{write} \lhd \mathsf{f} \lhd \mathsf{fetch} \lhd \mathsf{return} \circ \mathsf{out}\end{aligned}$$

Note that we do *not* use the default implementation for outIn_α, which in this case would use read instead of fetch. As explained in Section 5.3, fetch immediately frees a block after reading it. By using fetch instead of read, we get the effect that all modifications to the persistent data structure are *mutable* operations. After a modification finishes the old structure is no longer available.

With the OutIn instance we can now also specialize modification functions such as insert to work on the persistent storage:

$$\begin{aligned}&\mathsf{insert_P} :: \mathsf{Int} \rightarrow \mathsf{Int} \rightarrow \mathsf{Tree_P\ Int\ Int} \rightarrow \mathsf{Heap}\ (\mathsf{Tree_P\ Int\ Int})\\&\mathsf{insert_P} = \mathsf{insert}\end{aligned}$$

Similar to produce and consume, we define a function modify that applies a given modifier to the tree pointed at by the pointer in the root block, and stores the resulting tree pointer in the root block once more:

$$\begin{aligned}&\mathsf{modify} :: \mathsf{Binary}\ (\mathsf{f}\ (\mu\ \mathsf{f})) \Rightarrow (\mu\ \mathsf{f} \rightarrow \mathsf{Heap}\ (\mu\ \mathsf{f})) \rightarrow \mathsf{Heap}\ ()\\&\mathsf{modify\ c} = \mathsf{readRoot} \ggg \mathsf{c} \circ \mathsf{In} \ggg \mathsf{writeRoot} \circ \mathsf{out}\end{aligned}$$

Here is an example:

```
ghci> run "squares.db" (consume (lookupP 9))
Nothing
ghci> run "squares.db" (modify (insertP 9 81))
ghci> run "squares.db" (consume (lookupP 9))
Just 81
```

This is an interesting example because it shows three consecutive runs on the same database file. The second run modifies the binary tree on disk and stores the new root pointer in the root block. A lookup in the third command shows us the database file is updated.

6.3 Summary

This section we have combined the annotated recursion patterns and the basic heap operations to derive persistent data structures. By annotating the recursive datatypes with a pointer annotation we are able to store individual non-recursive nodes on their own block on the heap. The Out and In instances for the pointer type class read nodes from and write nodes to the blocks on disk.

The operations on persistent data structures are applied to the file based storage heap in the same way they are normally applied to the in-memory heap. By writing data structures as pattern functors and by abstracting from the recursion we can annotate the behaviour generically. There is no need to reflect over memory layout of the compiler runtime.

7. Discussion and future work

In this section, we discuss some subtleties of our approach. We also point out current shortcomings and topics for future work.

7.1 Laziness

The framework for working with annotated recursive datatypes uses type classes to associate functionality with creating and removing annotations at the recursive positions. These type classes have an associated context that allows annotating and un-annotating structures to have monadic effects. If the context is a *strict* context, the operations working on the recursive data structure become strict too. This strictness can have a severe and unexpected impact on the running time of the algorithms.

As an example, recall the lookup function on binary search trees as discussed in Section 4.1. Used with the identity annotation, the operation performs an in-memory lookup, traversing one path in the tree from the root to a leaf. If the tree is properly balanced, this corresponds to a runtime of $O(\log n)$ where n is the size of the tree. However, if used with the pointer annotation from Section 6, the lookup function runs inside the Heap monad which is strict, because the underlying IO monad is strict. The strict bind operator for the Heap monad makes the $\mathsf{lookup_P}$ operation traverse the entire tree, i.e., to run in $\Theta(n)$. The same happens if we use the modification time or debug annotation.

Two possible solutions for this problem come to mind:

- We can let algebras be monadic. The recursion patterns then pass computations rather than precomputed results to the algebras. It becomes the responsibility of the algebra implementor to explicitly evaluate the inputs that are needed.
- We can try to ensure that the operations run in a *lazy monadic context*. When the context is lazy, the entire operations becomes lazy while the algebras remain pure.

We have adopted the second option: We build our recursion patterns on top of *lazy monads*. We make a type class that can be used to lift monadic computations to lazy computations:

$$\begin{aligned}&\textbf{class}\ \mathsf{Lazy\ m}\ \textbf{where}\\&\quad \mathsf{lazy} :: \mathsf{m\ a} \rightarrow \mathsf{m\ a}\end{aligned}$$

We make an instance for the IO monad by using unsafeInterleaveIO. This function delays IO operations until they are actually required, possibly discarding them if their results are never used:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Lazy\ IO}\ \textbf{where}\\&\quad \mathsf{lazy} = \mathsf{unsafeInterleaveIO}\end{aligned}$$

For monads that are already lazy, we can instantiate lazy to be the identity function.

A new catamorphism can be built that uses invokes the lazy method just before going into recursion:

$$\mathsf{lazyCata}_\alpha\ \phi = \mathsf{return} \circ \phi \lhd \mathsf{mapM}\ (\mathsf{lazy} \circ \mathsf{lazyCata}_\alpha\ \phi) \lhd \mathsf{out}_\alpha$$

The lazy catamorphism ensures that the monadic actions will only be performed when the algebra requires the results. The type context tells us this catamorphism is only applicable to monads that can be run lazily. We derive a new lookup function using $\mathsf{lazyCata}_\alpha$:

$$\mathsf{lookup\ k} = \mathsf{lazyCata}_\alpha\ (\mathsf{lookup_{ALG}\ k})$$

When we perform a lookup on the output of myTree_α – specialized to the debug annotation – we see a clear reduction in the number of steps needed to compute the answer:

```
ghci> lookup 4 it
("out",Branch 3 9 () ())
("out",Branch 4 16 () ())
Just 16
```

We have solved the laziness problem for the storage heap specifically by creating two separate heap contexts, a read-only context which uses lazy IO and a read-write context that uses strict IO. The pointer instance for the Out type class is now associated with the read-only context, the instance for the In type class is associated with the read-write context.

To avoid any problems regarding lazy IO, we strictly force the entire result values of consumer operations to ensure all side-effects stay within the Heap context and cannot escape. Our operations are now lazy on the inside but appear strict on the outside.

7.2 Other data structures

We have shown how to build a generic storage framework for recursive data structures. As running example, we used binary search trees, but the same technique can easily be applied to any regular datatype, i.e., all types that can be expressed as a fixed point of a functor in terms of μ_α.

Non-regular datatypes such as families of mutually recursive datatypes, nested datatypes [3] and indexed datatypes or generalized algebraic datatypes (GADTs) [17] cannot be expressed directly. However, it is known that many non-regular datatypes can be expressed in terms of a *higher-order* fixed point combinator [9, 17, 23] such as

newtype $\mu^{\mathsf{h}}\ \mathsf{f}\ \mathsf{ix} = \mathsf{In}^{\mathsf{h}}\ (\mathsf{f}\ (\mu^{\mathsf{h}}\ \mathsf{f})\ \mathsf{ix})$

We have extended our annotation framework to such a setting. Each of the constructions described in the paper can be lifted to the more complex scenario, but no code reuse is directly possible due to the more complicated kinds in the higher-order situation.

A more in depth report about persistent indexed datatypes is provided by Visser [31]. He shows how to represent finger trees [12], a nested data structure supporting efficient lookup and concatenation, as an indexed GADT and use the higher-order storage framework to derive a persistent finger tree. All the structural invariants we expect the finger tree to have are encoded using the datatype indices.

7.3 Sharing

The storage framework as described works for finite data structures. Finite data structures that use sharing can be stored on disk using our framework, but because sharing in Haskell is not observable, shared substructures will be duplicated in the heap. Storing shared values more than once can be a serious space leak for datatypes that heavily rely on sharing.

Solutions have been proposed to make sharing in Haskell observable [5, 10]. These solutions are often not very elegant, because they require some form of reflection on the internal machinery of the compiler runtime.

It would be a useful extension to our framework to allow designers of functional data structures to explicitly mark points at which sharing is possible. Sharing markers can limit the amount of data used to store data structures on disk and can even allow cyclic data structures to be saved in a finite amount of space. Note that a data structure with explicitly marked points of sharing fits nicely into our general framework of representing data structures as annotated fixed points.

The current storage framework without explicit sharing does not require any special form of garbage collection. The modification functions written with the help of the OutIn type class will automatically clean up old nodes that are no longer needed. The ability of explicit sharing would change this, a modification cannot blindly free old nodes because they might be shared with other parts of the data structure. The addition of explicit sharing requires support for garbage collection.

7.4 Concurrency

The current framework only allows sequential access to the persistent data structures. Concurrent access currently would most certainly cause undesirable effects. Parallel access to the same persistent data structure is a topic for future research. We could benefit from in-memory transactions systems like *software transactional memory* [11] to manage concurrent threads to manipulate the same structure. Transactional in-memory caches to persistent data structures have been shown useful before [6].

Another approach to concurrent access is making the data structures immutable. Using read instead of the fetch in the OutIn type class would yield a framework were modification functions copy the original structures. Different threads can now work on their own version of a data structure. To make this approach practically usable we need full sharing between different versions of the data structures and need a garbage collector to clean up versions that are no longer used.

8. Related work

8.1 Generic programming with fixed points

The idea of using fixed points and recursion patterns to express datatypes and operations on such datatypes is well-explored [1, 16, 21]. This approach to structuring data is also known as *two-level types* [24]. While the original motivation for taking this view was mainly to derive algorithms generically or calculate laws – such as fusion laws for optimisation purposes, fixed-point representations have also been used to modify datatypes in various ways.

A few examples: Garrigue [8] shows how writing datatypes in an open way enables adding extra functionality at a later point. Swierstra [26] presents a very polished approach to a similar problem tailored to Haskell. The Zipper data structure can be generically derived from a fixed-point view [13, 23]. Van Steenbergen et al. [29] show how to use generic programming with annotated fixed points to store source position information in abstract syntax trees. Chuang and Mu [4] explore an approach similar to our own, using fixed-point representations for storing data on disk in the context of OCaml.

Most recursion patterns we use are standard, except for the endo-apomorphism defined in Section 4.5. This pattern somewhat resembles a *futumorphism* [27]. Monadic folds have been described by Fokkinga [7].

8.2 Lazy IO

Lazy IO in Haskell has many associated problems. Pure code processing values origination from effectful computations can trigger side effects and technically behave as impure code. Kiselyov [18] describes iteratee-based IO as a solution for the lazy IO problem. Until now their approach has only been shown useful for linear IO system, like processing a file line by line. Iterators have a structure similar to algebras for list catamorphisms, it is not sure whether the iteratee approach is extensible to different functor types, like the tree base functor.

8.3 Persistent storage in Clean

In their paper *Efficient and Type-Safe Generic Data Storage*, Smetsers, Van Weelden and Plasmeijer [25] describe a generic storage framework for the programming language Clean. Similar to our storage framework, they aim at generically mapping functional data structures to a persistent storage on disk. Using *Chunks* – a concept similar to our storage heap – they are able to store individual parts of the data structures on the disk without the need for reading and writing the entire collection at once.

The major difference between their approach and ours is that they do not slice the data structure at the recursive points, but at the points where the actual element values are stored. This means that every record value is stored in its own chunk, while the entire data structure itself is stored in one single chunk. Updates of individual record values can now be performed efficiently without touching the entire collection, but for every structural change to the collection the chunk containing the data structure itself (the *Root chunk*) has to be read in and written back as a whole.

8.4 Happstack State

The *Happstack* [14] project consist of Haskell web server and a state framework. The state framework is called *Happstack-State*. It uses a record-based system in which users can add, delete, modify and retrieve records of data on a database file. The system uses Template Haskell meta-programming to automatically derive storage operations for custom datatypes. The derivation of operations only works for monomorphic types which severely breaks modularity. Happstack State only allows storing record values and does not allow using custom domain specific data structures.

9. Conclusion

The power of programming languages is often correlated with the functionality the standard libraries expose. Mainstream languages all provide their own set of data structures that the programmer can use to manage information in application memory. Using the same structures to manage information on external storage devices is generally not possible.

With this work, we provide the possibility to use functional data structures implemented in Haskell to manage information outside application memory. As a low-level storage we use a heap structure that stores blocks of binary data on a file on disk. The heap can grow and shrink on demand. Operations traversing persistent recursive data structures read or write non-recursive nodes from and to the heap level by level. The incremental behaviour keeps the algorithms efficient, the asymptotic running time on disk is equal to that in-memory.

Writing persistent data structures requires to abstract from recursion in both the definitions of the datatypes and the definitions of the operations. The algebras used as a description for the recursions operations remain pure and annotation-agnostic. Working with persistent data structures is not very different from working with normal data structures, although all operations need to be lifted to the monadic Heap context. Converting between in-memory and persistent versions of a data structure is easy.

We encourage writing recursive datatype definitions as pattern functors and operations in terms of recursion patterns. Datatypes written this way are open to annotation which can be exploited in a variety of ways.

Acknowledgments

We thank Chris Eidhof, Tom Lokhorst, José Pedro Magalhães and the anonymous referees for their helpful remarks.

References

[1] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, pages 28–115, 1998.

[2] D. Barry and T. Stanienda. Solving the Java object storage problem. *Computer*, 31:33–40, 1998.

[3] R. Bird and L. Meertens. Nested datatypes. In *Mathematics of Program Construction*, pages 52–67. Springer, 1998.

[4] T.-R. Chuang and S.-C. Mu. Out-of-core functional programming with type-based primitives. In *Practical Aspects of Declarative Languages*, 2000.

[5] K. Claessen and D. Sands. Observable sharing for functional circuit description. In *In Asian Computing Science Conference*, pages 62–73. Springer, 1999.

[6] A. G. Corona. TCache: A transactional data cache with configurable persistence, 2009. `hackageDB: TCache`.

[7] M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical report, Memoranda Informatica 94-28, University of Twente, 1994.

[8] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*, 2000.

[9] N. Ghani and P. Johann. Initial algebra semantics is enough! In *Typed Lambda Calculus and Applications*, number 4583 in LNCS, pages 207–222, 2007.

[10] A. Gill. Type-safe observable sharing in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2009.

[11] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*, pages 48–60. ACM, 2005.

[12] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.

[13] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. In *Mathematics of Program Construction*, LNCS, pages 148–174. Springer, 2002.

[14] A. Jacobson. HAppS-State: Event-based distributed state, 2009. `hackageDB: HAppS-State`.

[15] P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43:2002, 2001.

[16] P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *Principles of Programming Languages*, pages 470–482. ACM, 1997.

[17] P. Johann. Foundations for structured programming with GADTs. In *Principles of Programming Languages*, pages 297–308, 2008.

[18] O. Kiselyov. iteratee: Iteratee-based I/O, 2009. `hackageDB: iteratee`.

[19] L. Kolmodin and D. Stewart. binary: Binary serialisation for Haskell values using lazy ByteStrings, 2009. `hackageDB: binary`.

[20] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5): 413–424, September 1992.

[21] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[22] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *ACM symposium on Theory of computing*, pages 137–142. ACM, 1972.

[23] A. Rodriguez Yakushev, S. Holdermans, A. Löh, and J. Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *International Conference on Functional Programming*, pages 233–244. ACM, 2009.

[24] T. Sheard. Generic unification via two-level types and parameterized modules. In *International Conference on Functional Programming*, pages 86–97. ACM, 2001.

[25] S. Smetsers, A. van Weelden, and R. Plasmeijer. Efficient and type-safe generic data storage. In *Workshop on Generative Technologies*, Budapest, Hungary, April 5 2008. Electronic Notes in Theoretical Computer Science.

[26] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

[27] T. Uustalu and V. Vene. Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica*, 10:5–26, 1999.

[28] T. van Noort, A. Rodriguez, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *Workshop on Generic Programming*, pages 13–24. ACM, 2008.

[29] M. van Steenbergen, J. P. Magalhães, and J. Jeuring. Generic selections of subexpressions. In *Workshop on Generic Programming*. ACM, 2010.

[30] V. Vene and T. Uustalu. Functional programming with apomorphisms (corecursion). In *Nordic Workshop on Programming Theory*, 1998.

[31] S. Visser. A generic approach to datatype persistency in Haskell. Master's thesis, Utrecht University, 2010. URL `http://github.com/sebastiaanvisser/msc-thesis/downloads`.