A Generic Approach to Datatype Persistency in Haskell

Sebastiaan Visser

MSc Thesis

May 6, 2010

INF/SCR-09-60

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

Daily Supervisor:
dr. A. Löh
Second Supervisor:
prof. dr. J.T. Jeuring

Abstract

Algebraic data types (ADTs) are a powerful way to structure data in Haskell programs, functions can be used to process values of these ADTs. When dealing with a large collection of data that does not fit into computer memory at once, tricks have to be used to make the data persistent on external storage devices. There are two important properties that frameworks for data persistence at least should have. First of all it should allow incremental access to parts of the data. By not requiring the entire data structure to be read from and written to disk at once the tool will scale well to large amounts of data. The second important property is that users of the system should not be bothered with the fact that the actual storage is located outside the application memory. A system that is truly transparent to the user will better fit the functional paradigm and therefore more easily be adapted. These two properties are essential for a persistence framework in order to be useful in practice. This document proposes a new persistence framework for Haskell that uses purely functional data structures to manage long lived data on disk. It projects both the data types and the algorithms working on these data types to a persistent storage. By not using any external database tools the system remains lightweight and does not compromise the functional paradigm.

Contents

Chapter 1

Introduction

Management of long-lived data is an essential ingredient of a large amount of modern computer programs. Over the past decades of software technology, plenty of systems have been developed to store information outside the process space of a running application. Examples of such systems are structured file systems, relational databases management systems[Cod70] (RDBMSs), XML document stores, key/value databases, and many more. All of these tools allow for some form of structuring the data. For example, file systems allow for a hierarchical view on data using directories and files, relational database management systems use a tabular layout in which data can be structured using – rows and columns, and key/value stores build up finite mappings. Giving structure to data greatly helps efficiently manipulating data. Unfortunately, the data structures used by such systems do not always match the data structures used inside the computer program that uses the data. The result of this structure mismatch is that for many computer programs code has to be written to convert the structure of data when writing it to a long-lived storage and when reading it back again.

Over the years many solutions have been proposed for this problem, for many different contexts. For example, for most object-oriented (OO) programming languages there exists Object-Relational Mappers[BS98] (ORMs) that allow for a transparent mapping between objects and tables within a relational databases. ORM tools exploit the structural properties of both the table based RDBMSs and objects in an object-oriented language to derive an automated mapping to the database. Automating the conversion step can greatly save time in the development process.

Unfortunately, an object-relational mapping is not applicable to functional programming languages like Haskell. Haskell uses algebraic datatypes to structure data, not objects described by classes. There are currently several ways of making algebraic datatypes in Haskell persistent on an external storage devices, but unfortunately all of these methods have some limitations. We will now discuss three commonly used techniques for data persistence in Haskell and explain the limitations of these approaches.

- **Relational databases.** There are several packages available for Haskell that use connections to existing relational database management systems to store values outside of application memory. Either using manually written conversions functions or via a generically derived conversion, values of arbitrary types will be mapped to database rows. In the related work section 7.2 some examples can be found of database connectors for Haskell.

  Unfortunately this mapping from algebraic datatypes to the table based layout of RDBMSs, without losing any structural information, tends to be rather hard and inefficient. Because the mapping from complex hierarchical Haskell data structures to a table based database system is hard, most database connectors only support a restricted set of types to be marshalled. This observation leads to the conclusion that writing a Haskell data model that describes the structure of an existing relational database is often very easy, but *using a relational database to store the values of the data model of an existing Haskell program can be rather hard.*

- **Key/value storage.** Similar to the connectors to relational databases are the mappers to key/value based storage systems. These systems provide an interface similar to the Haskell *Map* datatype, a finite mapping from keys to values. This interface can be very useful when managing a large amount of records that can be easily identified by a single key. Lookup of records by key can often be done very efficiently. Example of such key/value stores can be found in the related work section 7.3.

  While key/value stores can be useful, and are often easy to use, they have some limitations. *Key/Value stores only allow one data structure to be used, a finite mapping between keys and value.*

- **Textual and binary serialization.** Another possibility to store values of arbitrary datatypes outside the application memory is to serialize the entire value to a textual or binary representation. This representation can be written to and read from disk at once.

  The infamous Haskell *Show* and *Read* type classes are primitive examples of such a tool. These classes are used to print and parse textual representation of Haskell values and can be derived for most values generically by the compiler. More advanced tools exist that use binary serialization to perform the same trick more space and time efficient.

Some of the libraries are very fast and make use the types of values to prevent creating to much administrative overhead when saving the binary data.

The big disadvantage of these libraries is that values can only be written and read at once. Due to this property, these techniques do not scale well when dealing with very large amount of data.

1.1 Contributions

So there are several ways in Haskell to save the application data on a persistent storage like a hard disk. Unfortunately, all of these approaches have limitations. In this document we will describe the design and implementation a new framework for data persistency in Haskell. The storage framework this document describes has the following properties.

- **Pure Haskell** The framework is written entirely in Haskell. No connections to existing database tools are used. Writing the framework in pure Haskell has some important advantages. First of all the system remains lightweight and easily deployable in an existing Haskell environment. No foreign function interfaces to existing c-libraries are needed. Secondly, by writing the entire system in Haskell we are allowed to exploit the full power of the type system to guide or implementation. The Haskell type system allows us to both make sure our implementation is correct and helps us designing an interface to the end-users.

- **File based.** The system uses a technique for binary serialization to convert Haskell values to a stream of bytes. This stream of bytes is stored in a file based storage heap on disk. The file based heap can contain multiple blocks of binary data and can grow and shrink on demand.

- **Generality.** The framework can be used to store values of arbitrary Haskell datatypes. This includes container data structures like lists, binary trees, finger trees, abstract syntax trees, etc, but it also includes values of simpler types like integers, strings, *Maybe*s or values of any other algebraic datatype. Generic programming techniques are used to automatically derive code for binary serialization and deserialization of Haskell values.

- **Incrementality.** The storage framework allows incremental access to the persistent data stored in the file based heap. Operations on a persistent data set – like updates, insertions and queries – can be performed without reading the entire data store into memory. Most, if not all, Haskell container data structures are represented by recursive datatypes. The framework stores all non-recursive nodes of a recursive datatype in a separate block on disk and connects individual nodes together using pointers. By slicing larger data structures in pieces we can make sure fast incremental access to the data is possible. The amount of read and write actions are minimized as much as possible, because disk access is significantly slower compared to memory access. Incremental access is an important feature for writing programs that scale well to large data sets.

- **Three layer.** The framework can conceptually be separated in three different layers of functionality.

  1. The lowest layer is the *persistence layer* that allows values of Haskell datatypes to be stored on disk. The persistence layer makes sure recursive datatypes are sliced into pieces and stored in separate blocks on the storage heap.
  2. The second layer is the *data layer* and contains the persistent versions of the recursive container data structures. The recursive data structures are written in a special way that makes sure the *persistent layer* is able to lift them to the storage layer.
  3. The third and top-most layer is the *user layer*. This layer is the interface to the developer of the application requiring a persistent data store. The developer can choose a data structure that fits her needs and use high level operations to manipulate the data.

- **Transparency.** The framework is transparent to both the developer and the designer of the data structures that will be made persistent.

  When writing a persistent version of a data structure, no knowledge of the inner workings of the persistence framework is needed. The recursive datatypes and operations on the datatypes are written in pure Haskell code and are agnostic of the persistence layer. When written in the correct way, the pure operations can automatically be lifted to work on the block based storage heap instead of working in application memory.

  Manipulation of the persistent data structures by the application developer is just like manipulating a regular in-memory data structure. When performed operations on the data structure all I/O actions to the file heap are abstracted away. The only observable difference is that all operations are lifted to a *monadic* computational context. The monadic context allows the framework to transparently convert the operations to work on the heap.

With these six properties we have implemented a Haskell storage framework that is both lightweight, fast, easy to extend and easy to use.

1.2 The source code

This document describes the design and implementation of a generic storage framework for the functional programming language Haskell. The framework is implemented as a Haskell library and which is called *Islay*[1]. The source code of the library is available online on:

`http://github.com/sebastiaanvisser/islay`

Additionally, the sources of this document can be found:

`http://github.com/sebastiaanvisser/msc-thesis`

At the moment of writing the library is only a prototype and not in release-worthy state. When the library matures it will eventually be released on the public Haskell packages storage *Hackage*. This documentation is a description of the ideas and techniques from this library, not of the actual implementation. This means that certain parts of the code and examples may not correspond to the library exactly.

1.3 Motivating example: spatial indexing

The advantage of relational databases is the generality when it comes to storing data. Using (or misusing) the generic table based layout of an RDMS will almost always ensure that your application data can be saved in *some* structure. Unfortunately it is not always easy to perform fast queries over data when the structure of information in the table based layout does not fit the algebraic layout of the original algebraic datatype well.

To illustrate this disadvantage, consider an example application that stores a mapping from two-dimensional geometrical coordinates to business relations. Storing such a mapping in a database is very simple and does not take a very complicated structure. The problem arises when you want to perform efficient spatial queries over the data, like getting the n nearest business relations to the city center of Utrecht. Efficient SQL queries that perform such specific tasks might be very hard to write.

[1]After the Scottish island of Islay, the source of some very fine whiskys.

On the other hand, there are several data structures very capable of performing such spatial queries very efficiently. A quadtree is a domain specific data structure specialized for efficient indexing of spatial information. Quadtrees are specializations of multidimensional search trees[Ben75]. Elements inside a quadtree are saved and indexed based on their geometrical coordinates which results in very efficient spatial lookups. Finding the k elements nearest to a specific elements using a quadtree can be done in not more than $O(k \log (n))$ time. Performing the same task with the same asymptotic running time using SQL queries on an table based RDMS is very difficult and probably requires a lot of knowledge about the internals of the database.

Storing data is something most databases do very well, performing efficient domain specific queries over a collection of elements is probably best done using a specialized data structure. Storing a large collection of data outside application memory and still being able to perform specialized queries over the data is still an unsolved problem in most programming languages and accompanying storage libraries.

The framework described in this document solves the problem of persistent domain specific data structures by separating the concerns of persistent storage and that specialized data structures. By projecting the internal structure of container datatypes to an isomorphic variant on disk specialized algorithms will still be applicable with the same time and space complexities. The framework enables developers of container libraries to focus on the internal structure and allows for writing efficient algorithms without worrying about the internals of the storage layer.

1.4 Overview

This section gives a brief overview of the next chapters.

In chapter 2 we explain how to use an fixed point combinator to obtain control over the recursive positions of recursive data structures. By parametrizing recursive datatypes with a type parameter that is used at the recursive positions, the users of the datatype are allowed to change the values stored at these positions. The fixed point takes a datatype with a type parameter for the recursion and instantiates this parameter with its own fixed point, making the datatype truly recursive again. By replacing the regular fixed point combinator with an annotated fixed point combinator we are able to store additional information in the recursive positions of a recursive datatype. As an example, in section 2.2 we show how to use an annotated fixed point combinator to build an annotated binary tree datatype. We show that using an identity annotation at the recursive positions gives us back a structure isomorphic to a regular binary datatype. In section 2.3 we introduce two Haskell type classes that can be used to wrap and unwrap constructors of recursive datatypes with an annotation variable. These type classes allow us to associate functionality with the annotation variables stored in the recursive positions. As we will see in chapter 5, annotations allow us to lift recursive data structures to a storage heap.

Working with annotated recursive data structures on itself is not easy. All the operations working on the datatypes need to wrap and unwrap the annotations every time a recursive positions is touched. In chapter 3 we solve this problem by writing operations that abstract away from recursion. We show how to write operations on recursive data structures as algebras and coalgebras, the (co)algebras can be interpreted generic traversal functions. In section 3.1 and 3.2 we both define an annotated paramorphism and apomorphism function. The paramorphism uses an algebra to destruct an annotated data structure to some result value, thereby using the annotation type classes to unwrap the recursive values out of the annotations. The apomorphism uses a coalgebra to produce an annotated data structure from some seed value, thereby using the annotation type classes to wrap the recursive values in new annotations. Paramorphisms and apomorphisms can be used to respectively destruct and construct recursive data structures. In section 3.3 and 3.4 we define both annotated endomorphic paramorphisms and endomorphic apomorphisms that allow us to modify existing recursive data structures. In section 3.5 we show how to compose multiple algebras into one using the Haskell *Applicative* type class. Composing algebras can simply building algebraic operations. The traversals described in this chapter all work on annotated structures, when the wrap and unwrap functions associated with the annotation requires a strict computational context the running time of the operations can be negatively influenced. In section 3.6 we solve the strictness problem by enforcing the traversals to be as lazy as possible.

In chapter 4 we show the file based heap that forms the low level storage system of the framework. Section 4.1 describes the block based layout of the heap and how we can use offset pointers to refer to blocks of data. We describe the basic operations of the heap: reading data from a block, writing data to a block, allocation a new block, and freeing an existing block. In section 4.5 we build three functions that help use to perform heap operations that require a fixed root node as a starting point. In 4.6 we show how to run heap operations against a binary file. All heap operations work inside a heap context, a monadic computational context that abstracts all low level I/O operations.

In chapter 5 we show how we can make the pointer type from the storage heap from chapter 4 an instance of the annotation type classes from chapter 2. Wrapping a pointer annotation represents writing a single node of a recursive data structure to the heap. Unwrapping an annotation represents reading a value from the heap. In section 5.3 we show how the pointer annotation can be used to derive a persistent binary tree. We specialize the generic binary tree operations to work with the pointer annotation in the heap context. Using these operations in the heap context results in a binary that lives on disk instead of in application memory.

The storage framework from chapter 5 can be used to build persistent variant of recursive data structures. Because the recursive data structures now live on disk, they survive the lifetime of a single program invocation. Unfortunately this framework only works for regular recursive datatypes. There is a class of recursive Haskell datatype that can not be used by this framework. In chapter 6 we extend the framework to allow making indexed recursive data types persistent on disk. We extend the annotation framework and the generic traversals to work higher order datatypes. In section 6.2 we introduce the finger tree data structure. We write down the definition of the finger tree as an indexed generalized algebraic datatype. We show how to write operations on the finger tree as higher order algebras.

In chapter 7 we give a listing of work related to this project. We show some existing library for data storage in Haskell and explore some topics that influence this work or can be an inspiration for future work. In chapter 8 we give a list of possible extensions of this framework. Lots of functionality can be added and lots of design decisions can be reconsidered. This chapter states the most obvious of topics for possible future work. In chapter 9 we wrap up with a conclusion.

Chapter 2

Annotated fixed points

todo: intro

todo: status: reviewed

2.1 Fixed points

Most container datatypes in Haskell are written down with explicit recursion. An example of a container type using explicit recursion is the following binary tree datatype. This binary tree stores both a value and two explicit sub-trees in the branch constructor, empty trees are indicated by a leaf.

```
data Tree = Leaf | Branch Int Tree Tree
```

In figure 2.1 we see an example of binary tree with four values.

To gain more control over the recursive positions of the datatype we can parametrize the binary tree with an additional type parameter used at the recursive positions. Not the tree datatype itself, but the users of the datatype may now decide what values to store as sub-trees. We call this new datatype $Tree_f$, the tree functor.

```
data Tree_f f = Leaf | Branch Int f f
  deriving Show
```

To get back a binary tree that is isomorphic to our original binary tree, in that it stores actual sub-trees at the recursive points, we can use an explicit fixed point combinator at the type level. This combinator, conventionally called μ, takes a type constructor of kind $\star \to \star$ and parametrizes this type with its own fixed point.

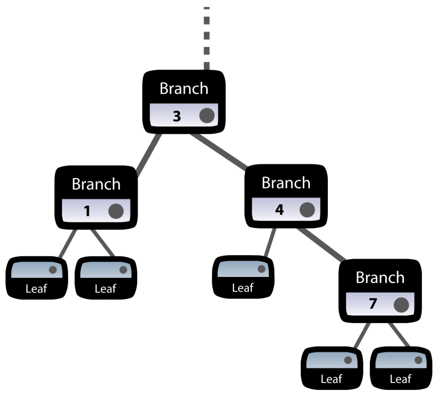

Figure 2.1: An example of a binary tree.

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

By applying the fixed point combinator to the tree functor we get a back a true binary tree again, with real sub-trees at the recursive positions.

```
type Tree = μ Tree_f
```

We will call datatypes that abstract away from recursion using an additional type parameter *open recursive datatypes*.

To make it easier to deal with the recursive structure of the binary tree we can make the tree functor an instance of Haskell's *Functor* type class. The functorial *fmap* takes the input function and lifts it to be applied against the sub-structures of the binary tree.

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

Besides *Functor* Haskell has two additional type classes that help with generic traversals over container datatypes. These are the *Foldable* and *Traversable* type classes[1]. The *Foldable* type class allows us to reduce an entire structure into a single value using some *Monoid* operation.

```
instance Foldable Tree_f where
  foldMap _ Leaf           = ∅
  foldMap f (Branch _ l r) = f l ⊕ f r
```

[1]Note that these type class instances are very simple and mechanically derivable. The GHC Haskell compiler version 6.12.1 and above is able to derive the instances for *Functor*, *Foldable* and *Traversable* for you automatically.

The *Traversable* type class, which requires *Foldable* as its super class, allows a generic traversal over a structure while performing an action for each element. The actions performed are *Applicative* or sometimes *Monad*ic computations. The *Traversable* instance for our binary tree example is a straightforward preorder traversal. The actions are written down using idiom brackets. [MP] show how idiom brackets can be used for effecful applicative programming.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Traversable}\ \mathit{Tree}_f\ \textbf{where} \\
\quad \mathit{traverse}\ \_\ \mathit{Leaf} \qquad\qquad\ = (\!|\ \mathit{Leaf}\ |\!) \\
\quad \mathit{traverse}\ f\ \ (\mathit{Branch}\ v\ l\ r) = (\!|\ (\mathit{Branch}\ v)\ (f\ l)\ (f\ r)\ |\!)
\end{array}
$$

Having instances of the *Traversable* class around is very useful, because it allows us to use the generic version of the Prelude's *mapM* function. This function enables us to *fmap* a monadic action over a structure and transpose the result.

$$\mathit{mapM} :: (\mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow (\alpha \to m\ \beta) \to f\ \alpha \to m\ (f\ \beta)$$

The *mapM* function can be used to perform a very lightweight form of generic programming.

2.2 Annotations

In the previous section we worked out some basic building blocks that can be useful when working with container datatypes which are explicitly parametrized with the recursive structures. But why would it be useful to abstract away from recursion in the first place? This section will show how we can store additional information at the recursive positions of open recursive datatypes using an annotated fixed point combinator.

First we introduce a new fixed point combinator that optionally stores an annotation over a container datatype instead of a datatype directly. This type level fixed point combinator is called μ_α. [2] Throughout this document the *alpha* postfix will be used to indicate that a type or a function is annotation aware. The μ_α combinator has two constructors, one that stores an annotation over a structure f and one that stores a plain unannotated f, with possibly annotated sub-structures.

$$
\begin{array}{l}
\textbf{data}\ \mu_\alpha\ \alpha\ f = \\
\qquad \mathit{In}_\alpha\ \{\mathit{out}_a :: (\alpha\ f)\ (\mu_\alpha\ \alpha\ f)\} \\
\quad\ |\ \mathit{In}_f\ \{\mathit{out}_f :: \quad f \quad (\mu_\alpha\ \alpha\ f)\}
\end{array}
$$

[2] The μ_α **newtype** might feel redundant at first sight, because we could as well just parametrize the original μ with an annotated structure $(\alpha\ f)$, yielding the same result. From the usage of the μ_α it has become clear that expressing the more specific fixed point μ in terms of the more general μ_α helps us to more easily reuse functionality later on.

Note the kind of the annotation variable α, the annotation is applied to the original container type f which has kind $\star \rightarrow \star$. Because the annotation itself applied to the container type f needs to have the same kind as f, the variable α has kind $(\star \rightarrow \star) \rightarrow (\star \rightarrow \star)$.

It is now very easy to define a fully annotated binary tree by applying the annotated fixed point combinator to the tree functor.

$$\textbf{type}\ \mathit{Tree}_{\alpha}\ \alpha = \mu_{\alpha}\ \alpha\ \mathit{Tree}_{f}$$

In figure 2.2 we see an example of binary tree that uses an annotated fixed point to tie the recursive knot.

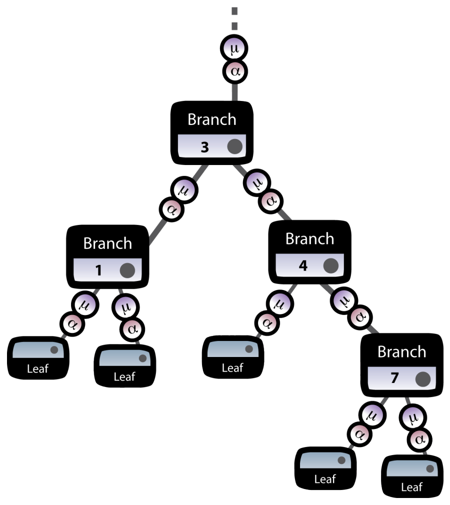

Figure 2.2: An example of an annotated binary tree.

We now introduce the identity annotation, called Id, that stores no additional information but just encapsulates the underlying container type.

$$\textbf{newtype}\ \mathit{Id}\ f\ \alpha = \mathit{Id}\ \{\mathit{unId} :: f\ \alpha\}$$

The identity annotation can be used to get back the regular fixed point combinator defined in the previous section by plugging it into a μ_{α}. Because the identity annotation stores no additional information we call a $\mu\ f$ structure an unannotated or plain structure.

$$\textbf{type}\ \mu\ f = \mu_{\alpha}\ \mathit{Id}\ f$$

Working with a fully annotated structure using the μ_{α} combinator or working with a plain structure using the μ combinator both require all substructures to be surrounded by an additional In constructor. To make the usage of plain binary trees more easy we create a *Tree* type synonym and two smart constructors: *leaf* and *branch*.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Tree} = \mu\ \mathit{Tree}_f \\
&\mathit{leaf} :: \mathit{Tree} \\
&\mathit{leaf} = \mathit{In}_\alpha\ (\mathit{Id}\ \mathit{Leaf}) \\
&\mathit{branch} :: \mathit{Int} \to \mathit{Tree} \to \mathit{Tree} \to \mathit{Tree} \\
&\mathit{branch}\ v\ l\ r = \mathit{In}_\alpha\ (\mathit{Id}\ (\mathit{Branch}\ v\ l\ r))
\end{aligned}
$$

The annotated fixed points can be used to store arbitrary pieces of data at the recursive positions of a recursive structure. To illustrate this using something more interesting than the identity annotation we annotate a binary tree with local modification times. In the following example every sub-structure will be surrounded with an annotation that stores a Haskell *LocalTime*, which might be filled in with the last time a sub-structure was modified.

$$
\begin{aligned}
&\textbf{data}\ \mathit{TimeAnn}\ f\ \alpha = \mathit{TA}\ \mathit{LocalTime}\ (f\ \alpha) \\
&\textbf{type}\ \mathit{TimedTree} = \mu_\alpha\ \mathit{TimeAnn}\ \mathit{Tree}_f
\end{aligned}
$$

2.3 Annotation associated functionality

In the previous section we have shown how to store arbitrary pieces of information at the recursive positions of a datatype. In this section we will show how to associate functionality with these annotations. For every annotation type we will describe how to obtain an annotation for a previously unannotated node and how to get a node out of a fully annotated structure. We create one type synomym for the process of putting a structure inside an annotation an one for getting a structure out of an annotation. We call an In_α function a producer function and a *Out* function a query function.

$$
\begin{aligned}
&\textbf{type}\ \mathit{In}\ \ \ \alpha\ f\ m = f\ (\mu_\alpha\ \alpha\ f) \to m\ (\ \ \mu_\alpha\ \alpha\ f) \\
&\textbf{type}\ \mathit{Out}\ \alpha\ f\ m = \ \ \ \mu_\alpha\ \alpha\ f\ \ \ \to m\ (f\ (\mu_\alpha\ \alpha\ f))
\end{aligned}
$$

As the type signature shows, a producer will take a node with fully annotated sub-structures and introduces a new annotation for this node making it a fully annotated structure again. The function might run in some – possibly monadic – context *m* when this is required for the annotation. The type signature for queries shows that it will take a fully annotated structure and will use the annotation to give back an unannotated node with the sub-structures still fully annotated. Like the producer, this functions can also run in some context *m*.

Two type classes are used to associate specific functionality to annotations. For producers this class is called Ann_{In}, for queries this class is called Ann_{Out}. Both type classes contain a single function with the type signature as defined above. The first parameter of the type class, α, is the annotation type, the second parameter, f, is the structure to annotate, the third, m, is the context it may run in.

```
class (Traversable f, Monad m) ⇒ Ann_In α f m where
  ann_In :: In α f m
class (Traversable f, Monad m) ⇒ Ann_Out α f m where
  ann_Out :: Out α f m
```

Making an annotation type an instance of these type classes means we can come up with an annotation for a structure and we can get back a structure from an annotation again. Note that the *Traversable* and the *Monad* [3] classes in the context are not strictly necessary super classes here. These constraints only help to prune the contexts when using the Ann_{Out} and Ann_{IO} classes, because then *Traversable* and *Monad* are both implied.

Now we can make the identity annotation an instance of both the Ann_{In} and Ann_{Out} type classes. We just unpack or pack the annotation and strip off or introduce the In_α constructor. For the In_f case we do not need to do any work.

```
instance (Traversable f, Monad m) ⇒ Ann_Out Id f m where
  ann_Out (In_α (Id f)) = return f
  ann_Out (In_f     f)  = return f


instance (Traversable f, Monad m) ⇒ Ann_In Id f m where
  ann_In = return ∘ In_α ∘ Id
```

Although redundant in the general case, for possible future optimizations we also introduce a type class for the modification of a sub-structure, called Ann_{IO}. The ann_{IO} function is used to apply a function over a single node within a fully annotated structure. There is a default implementation available which is just the Kleisli composition (denoted by •) of the query, the function, and the producer.

```
type InOut α f m = (f (μ_α α f) → m (f (μ_α α f)))
                 → (   μ_α α f  → m (   μ_α α f))
```

[3] In all the examples that follow we assume that the occurrence of *Monad* in a type context also assume the existence of an *Applicative* instance. Although this assumption is not strictly the case in Haskell it is valid in theory and saves us some typing.

$\textbf{class}\ (\mathit{Ann}_{\mathit{Out}}\ \alpha\ f\ m, \mathit{Ann}_{\mathit{In}}\ \alpha\ f\ m) \Rightarrow \mathit{Ann}_{\mathit{IO}}\ \alpha\ f\ m\ \textbf{where}$
$\quad \mathit{ann}_{\mathit{IO}} :: \mathit{InOut}\ \alpha\ f\ m$
$\quad \mathit{ann}_{\mathit{IO}}\ f = \mathit{ann}_{\mathit{In}} \bullet f \bullet \mathit{ann}_{\mathit{Out}}$

For the identity annotation we just use the default implementation for $\mathit{Ann}_{\mathit{IO}}$.

$\textbf{instance}\ (\mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow \mathit{Ann}_{\mathit{IO}}\ \mathit{Id}\ f\ m$

Now that we have defined both annotated fixed points and a type class to associate functionality with annotations we can create two smart constructors to simplify creating annotated binary trees manually.

$\mathit{leaf}_{\alpha} :: \mathit{Ann}_{\mathit{In}}\ \alpha\ \mathit{Tree}_f\ m \Rightarrow m\ (\mathit{Tree}_{\alpha}\ \alpha)$
$\mathit{leaf}_{\alpha} = \mathit{ann}_{\mathit{In}}\ \mathit{Leaf}$
$\mathit{branch}_{\alpha} :: \mathit{Ann}_{\mathit{In}}\ \alpha\ \mathit{Tree}_f\ m \Rightarrow \mathit{Int} \to \mathit{Tree}_{\alpha}\ \alpha \to \mathit{Tree}_{\alpha}\ \alpha \to m\ (\mathit{Tree}_{\alpha}\ \alpha)$
$\mathit{branch}_{\alpha}\ v\ l\ r = \mathit{ann}_{\mathit{In}}\ (\mathit{Branch}\ v\ l\ r)$

2.4 Multi-level annotation

In the previous chapter we have introduced how to wrap and unwrap a single level of a fully annotated structure. In this chapter we will introduce two additional functions that allows us to perform multi-level wrapping and unwrapping of annotations.

First we define the function $\mathit{fully}_{\mathit{Out}}$ that recursively unwraps all annotations from the top of an annotated strucuture. Only unwrapping annotations at the top means this function *assumes* that once it finds an unannotated node the functions stops. An unannotated node is indicated by the use of the In_f constructor in the fixed point.

$\mathit{fully}_{\mathit{Out}} :: (\mathit{Traversable}\ f, \mathit{Ann}_{\mathit{Out}}\ \alpha\ f\ m) \Rightarrow \mu_{\alpha}\ \alpha\ f \to m\ (\mu_{\alpha}\ \alpha\ f)$
$\mathit{fully}_{\mathit{Out}}\ (\mathit{In}_{\alpha}\ \alpha) = \mathit{ann}_{\mathit{Out}}\ (\mathit{In}_{\alpha}\ \alpha) \rightarrowtail \mathit{fmap}\ \mathit{In}_f \circ \mathit{traverse}\ \mathit{fully}_{\mathit{Out}}$
$\mathit{fully}_{\mathit{Out}}\ (\mathit{In}_f\ f)\ \ = \mathit{return}\ (\mathit{In}_f\ f)$

The dual function $\mathit{fully}_{\mathit{In}}$ performs the inverse process of $\mathit{fully}_{\mathit{Out}}$, it recursively annotated the top a (partially) unannotated structure. It recursively wraps all unannotated nodes in an annotation, when it finds a node that is already annotated it stops.

$\mathit{fully}_{\mathit{In}} :: (\mathit{Traversable}\ f, \mathit{Ann}_{\mathit{In}}\ \alpha\ f\ m) \Rightarrow \mu_{\alpha}\ \alpha\ f \to m\ (\mu_{\alpha}\ \alpha\ f)$
$\mathit{fully}_{\mathit{In}}\ (\mathit{In}_f\ f)\ \ = \mathit{traverse}\ \mathit{fully}_{\mathit{In}}\ f \rightarrowtail \mathit{ann}_{\mathit{In}}$
$\mathit{fully}_{\mathit{In}}\ (\mathit{In}_{\alpha}\ \alpha) = \mathit{return}\ (\mathit{In}_{\alpha}\ \alpha)$

When we assume the invariant that all the sub trees of an unannotated node do not contain any annotations, *fullOut* makes sure the entire structure will be unannotated. When we assume the invariant that all sub trees of an annotated node are fully annotated, $fully_{In}$ makes sure the entire structure will be annotated.

In the chapter **todo:** XXX about generic annotated traversals we will see that the $fully_{In}$ function will simplify writing algebras for both endomorphic paramorphisms and endomorphic apomorphisms.

2.5 Debug annotation

To more clearly demonstrate the usage of generic traversals over annotated structures in the next section we first introduce the *Debug* annotation. In contrast to the identity the debug annotation does have associated functionality. It will print out a trace of every node that gets *produced* or *queried*.

First we define the *Debug* datatype that is just a **newtype** similar to the identity annotation. No additional information is stored, the **newtype** is only used to associate specific actions to this annotation.

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

Now we create a little helper function that can print out a predefined prefix together with the some value and returns that same value again. Note that function does not directly run in the *IO* monad, but in some monad m for which there is a *MonadIO* instance, making it a bit more generally applicable.

```
printer :: (MonadIO m, Show β) ⇒ String → β → m β
printer s f =
  do liftIO (putStrLn (s ++ ": " ++ show f))
     return f
```

The Ann_{Out} instance for the *Debug* annotation justs unpacks the constructors and prints out the node that is queried, including the fully annotated sub-structures.

```
instance (Traversable f, MonadIO m, Show (f (μα Debug f)))
   ⇒      AnnOut Debug f m
  where annOut (Inα (D f)) = printer "ann0" f
        annOut (Inf     f ) = printer "ann0" f
```

The same trick can be done for the dual instance Ann_{In}. This function adds the In_α and D constructors and also prints out the node that is being produced.

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Traversable}\ f, \mathit{MonadIO}\ m, \mathit{Show}\ (f\ (\mu_\alpha\ \mathit{Debug}\ f))) \\
&\quad \Rightarrow \quad \mathit{Ann}_{\mathit{In}}\ \mathit{Debug}\ f\ m \\
&\quad \textbf{where}\ \mathit{ann}_{\mathit{In}} = \mathit{fmap}\ (\mathit{In}_\alpha \circ D) \circ \mathit{printer}\ \texttt{"annI"}
\end{aligned}$$

For the $\mathit{Ann}_{\mathit{IO}}$ we use the default implementation.

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Traversable}\ f, \mathit{MonadIO}\ m, \mathit{Show}\ (f\ (\mu_\alpha\ \mathit{Debug}\ f))) \\
&\quad \Rightarrow \quad \mathit{Ann}_{\mathit{IO}}\ \mathit{Debug}\ f\ m
\end{aligned}$$

In order to get the above class instances to work properly we additionally need a Show instance for our recursive structures. We represent the In_α constructor by surrounding recursive structures with triangular brackets.

$$\begin{aligned}
&\textbf{instance}\ \mathit{Show}\ ((\alpha\ f)\ (\mu_\alpha\ \alpha\ f)) \Rightarrow \mathit{Show}\ (\mu_\alpha\ \alpha\ f)\ \textbf{where} \\
&\quad \mathit{show}\ f = \texttt{"<"} +\!\!+\ \mathit{show}\ (\mathit{out}_a\ f) +\!\!+ \texttt{">"}
\end{aligned}$$

In the next chapters we will see how we can use the *Debug* annotation to print out debug traces of generic traversals over annotated structures. Printing out debug traces is just one example of what you can do with the annotation type classes. In section **todo:** ref to section we will show how to use the same annotation type classes to store and retrieve annotated structures to and from disk.

Chapter 3

Annotated Generic Traversals

In the previous chapter we have seen how to associate functionality with annotations. In this chapter will show how to write operations over annotated recursive data structures. We make sure that all operations we write are annotation unaware, which means we will use an existential annotation variable in the type signature of our operations. This existential makes sure the operations can be used in combination with all possible annotations.

Writing annotation-generic operations can only be done when the functions cannot touch the recursive positions of the annotated datatypes, because the recursive positions contain the annotations. We have to find a way to abstract away from recursion when writing our algorithms.

In this chapter we will use a well known functional programming technique for working with recursive datatypes. This technique has been explained by Meijer et al. in their paper *Functional programming with bananas, lenses, envelopes and barbed wire.*[MFP91] We will introduce *morphisms* to write operations that abstract away from recursion. We will implement both an annotation aware *paramorphism* and *apomorphism*. By writing *algebras* for these morphisms we will be able to destruct and construct recursive data structures without explicitly touching the recursive positions. By creating an endomorphic paramorphism and an endomorphic apomorphism we will also be able to update existing recursive structures.

In the last part of this chapter we show how to combine multiple algebras into one. This will allow us to perform multiple actions in a single tree traversal. We will also investigate the effect of traversals for annotations that work in a strict context on the running time of the operations.

3.1 Paramorphisms

We start out by implementing a *paramorphism*[Mee92], a bottom up traversal that can fold a recursive structure into a single value. A paramorphism is a generalization of the more commonly known *catamorphism*[MFP91]. The standard Haskell function *foldr*, which can be used to destruct a list to a result value, is an example of a catamorphism.

We first write down the type signature of the algebra for paramorphisms, we call this algebra Ψ_α.

$$\textbf{type}\ \Psi_\alpha\ \alpha\ f\ r = f\ (\mu_\alpha\ \alpha\ f \times r) \to r$$

This type signature describes an algebra that should be able to produce an value of type r from one single node containing both the fully annotated sub-structures *and* the recursive results of the paramorphic computation.

We now create a derived algebra type that hides the annotation variable inside an existential quantification. This makes explicit that the algebras cannot reason about the annotation.

$$\textbf{type}\ \Psi\ f\ r = \forall \alpha . \Psi_\alpha\ \alpha\ f\ r$$

An example of such an algebra is the function $contains_{alg}$ for binary trees. This algebra describes a recursive traversal over a binary tree that checks whether a certain integer value is included in the tree or not.

```
contains_alg :: Int → Ψ Tree_f Bool
contains_alg _ Leaf                    = False
contains_alg v (Branch c (_, l) (_, r)) =
  case v ‘compare‘ c of
    LT → l
    EQ → True
    GT → r
```

Note that because the $contains_{alg}$ algebra only uses the recursive sub-results, and not the original sub-structures, this algebra is actually a catamorphism, a special case of the more general paramorphism. Because all catamorphisms are paramorphisms this does not invalidate the example.

The paramorphism function performs a bottom up traversal over some *Traversable Functor* and for every node applies the algebra, the result of the algebra will be returned. The most generic version of this paramorphism within our framework is the $para^m_\alpha$ function. This function runs is some monadic context m and performs a traversal over some annotated structure $\mu_\alpha\ \alpha\ f$ using the Ann_{Out} type class to perform annotation specific queries.

$$
\begin{aligned}
&para^m_\alpha :: Ann_{Out}\ \alpha\ f\ m \Rightarrow \Psi_\alpha\ \alpha\ f\ r \to \mu_\alpha\ \alpha\ f \to m\ r \\
&para^m_\alpha\ \psi = return \circ \psi \bullet mapM\ (group\ (para^m_\alpha\ \psi)) \bullet ann_{Out} \\
&\quad \textbf{where}\ group\ f\ c = fmap\ ((,)\ c)\ (f\ c)
\end{aligned}
$$

From now on the $\binom{m}{\alpha}$ postfix will be used to indicate that a function requires a context and works on annotated structures.

The implementation of this generic paramorphism might seem a bit cryptic at first sight, this is due to its very generic behaviour. Quickly summarized this function performs a bottom-up traversal over a recursive structure like our binary tree. As input it receives a fully annotated structure and it uses the ann_{Out} function to unwrap a single node out of the annotation. The *Traversable* instance, which is an implicit super class of the Ann_{Out} class, allows us to use the *mapM* function to recursively apply the $para^m_\alpha$ function to the sub-structures. This recursive invocation is used to come up with the sub-results. The sub-results will be grouped together with the original sub-structures that these results are computed from. The original input node with these grouped results as the values will be passed into the algebra ψ. The algebra can now compute the result value for one level of the recursive computation, possible using the results of deeper traversals.

To illustrate the usage of the $para^m_\alpha$ function we apply it to the $contains_{alg}$ algebra and get back a true function that performs a containment check over a fully annotation binary tree.

$$
\begin{aligned}
&contains^m_\alpha :: Ann_{Out}\ \alpha\ Tree_f\ m \Rightarrow Int \to Tree_\alpha\ \alpha \to m\ Bool \\
&contains^m_\alpha\ v = para^m_\alpha\ (contains_{alg}\ v)
\end{aligned}
$$

We can easily test this function in the interactive environment of the GHC compiler. We first manually construct a binary tree and constrain this to the *IO* context and *Debug* annotation. While the binary tree is being constructed, using our previously defined smart constructors, the debug annotation prints out a trace of all nodes being produced.

```
ghci> join (branchA 3 <$> leafA <*> leafA) :: IO (TreeA Debug)
annI: Leaf
annI: Leaf
annI: Branch 3 <D Leaf> <D Leaf>
<D Branch 3 <D Leaf> <D Leaf>>
```

Now we can apply the $contains^m_\alpha$ function to the resulting binary tree and check for the existence of a *Branch* with value 3. While running this function the debug annotation prints out a trace of all sub-structures being queried.

```
ghci> containsMA 3 it
```

```
ann0: Branch 3 <D Leaf> <D Leaf>
ann0: Leaf
ann0: Leaf
True
```

Note that the paramorphic traversal is as strict as the context it runs in. This means that because the *Debug* annotation requires the *IO* monad the $contains_\alpha^m$ function becomes more strict then necessary. In section 3.6 we will describe a method to regain laziness for paramorphisms running in strict contexts.

The paramorphism we have defined above is generic in the sense that it works on structures with arbitrary annotations that run in an arbitrary context. When an annotation does not have any requirements about the type of context to run in, we can use the *Identity* monad to create a pure paramorphic traversal.

$$
\begin{aligned}
&para_\alpha :: (Ann_{Out}\ \alpha\ f\ Identity, Traversable\ f) \Rightarrow \Psi_\alpha\ \alpha\ f\ r \rightarrow \mu_\alpha\ \alpha\ f \rightarrow r \\
&para_\alpha\ \psi = runIdentity \circ para_\alpha^m\ \psi
\end{aligned}
$$

When we further restrict the annotation to be the identity annotation, we get back a pure paramorphism that works on plain unannotated structures.

$$
\begin{aligned}
&para :: Traversable\ f \Rightarrow \Psi_\alpha\ Id\ f\ r \rightarrow \mu\ f \rightarrow r \\
&para\ \psi = para_\alpha\ \psi
\end{aligned}
$$

To illustrate this pure paramorphism we apply it to the $contains_{alg}$ algebra and get back a pure *contains* function.

$$
\begin{aligned}
&contains :: Int \rightarrow Tree \rightarrow Bool \\
&contains\ v = para\ (contains_{alg}\ v)
\end{aligned}
$$

In this section we have shown how to build an annotation aware paramorphism, which can be applied to annotation-generic algebras. The $para_\alpha^m$ function is generic in both the annotation and the context the annotation requires. By only restricting the types we can derive operations that operate over the pure, in-memory variants of our data structures. In the next chapter will we do the same for *apomorphisms* which can be used to construct recursive data structures from a seed value.

3.2 Apomorphisms

Dual to the paramorphism is the *apomorphism*[VU98]. Where the paramorphism abstract away from recursion, the apomorphisms abstracts away from corecursion. Similarly, where paramorphisms use algebras to describe recursive operations, apomorphisms use coalgebras to describe corecursive operations. Apomorphisms are generalizations of the more well known *anamorphisms*. The standard Haskell function *unfold*, which can be used to create lists from a seed value, is an example of an anamorphisms.

The coalgebra for an apomorphism, called Φ_α, takes a seed value of some type s and must produce a node containing either a new seed or a new recursive structure.

$$\textbf{type}\ \Phi_\alpha\ \alpha\ f\ s = s \to f\ (s + \mu_\alpha\ \alpha\ f)$$

From the type signature of the Φ_α coalgebra it is obvious that it is dual to the Ψ_α algebra for paramorphisms. Paramorphisms destruct recursive structures to some result value r, apomorphisms construct recursive structures from some seed value s.

We now create a derived coalgebra type that hides the annotation variable inside an existential quantification. This makes explicit that the coalgebras cannot reason about the annotation.

$$\textbf{type}\ \Phi\ f\ s = \forall \alpha . \Phi_\alpha\ \alpha\ f\ s$$

Because the annotation variable α is not accessible to coalgebras that are written using the Φ type synonym, it can never produce annotation values. But the coalgebras is allowed to stop the corecursive traversal by using the R constructor to return a value of type $\mu_\alpha\ \alpha\ f$, an annotated structure. There are three ways the coalgebras can use the second component of the result sum type to stop the corecursion:

1. The first posibility for the coalgebra to stop the corecursive process is to return an unannotated structure using the R constructor. This can only be done by using the In_f fixed point constructor.
2. The coalgebra can *reuse* an existing annotated structure it receives as input. This can only work when the input is also an annotated structure, and this can only be the case when the seed type is exactly $\mu_\alpha\ \alpha\ f$. Apomorphisms that have a seed type equal to the result of the corecursion are called *endomorphic* apomorphisms, these will be discussed in section 3.4.
3. The third options is to wrap an existing annotated structure with one or more levels of unannotated nodes. This method is a combination of method 1 and method 2.

To illustrate the usage of coalgebras we define the function $fromList_{coalg}$ that describes how to create a balanced binary tree from an input list of integer values.

$$\begin{aligned}
&fromList_{coalg} :: \Phi\ Tree_f\ [Int] \\
&fromList_{coalg}\ [\,] \qquad = Leaf \\
&fromList_{coalg}\ (y : ys) = \\
&\quad \mathbf{let}\ l = take\ (length\ ys\ \text{`}div\text{'}\ 2)\ ys \\
&\quad\quad\ \ r = drop\ (length\ l)\ ys \\
&\quad \mathbf{in}\ Branch\ y\ (L\ l)\ (L\ r)
\end{aligned}$$

Note that because the $fromList_{coalg}$ coalgebra only produces new seeds using the L constructor, instead of directly producing sub-structures, it is actually an anamorphism. Because all anamorphisms are apomorphisms this does not invalidate the example.

Like the paramorphism we start with an apomorphism that corecursively generates an annotated structure in some, possibly monadic, context. We call this function apo_α^m. This apomorphism takes a coalgebra Φ_α and some initial seed value s and uses this to produce an annotated structure $\mu_\alpha\ \alpha\ f$.

$$\begin{aligned}
&apo_\alpha^m :: Ann_{In}\ \alpha\ f\ m \Rightarrow \Phi_\alpha\ \alpha\ f\ s \rightarrow s \rightarrow m\ (\mu_\alpha\ \alpha\ f) \\
&apo_\alpha^m\ \phi = ann_{In} \bullet mapM\ (apo_\alpha^m\ \phi\ \text{`}either\text{'}\ fully_{In}) \circ \phi
\end{aligned}$$

This apomorphism first applies the algebra ϕ to the initial seed value s to produce a new structure of type $f\ (s + \mu_\alpha\ \alpha\ f)$. This results has either a new seed or a recursive sub-structure in the sub-positions. When we encounter a new seed in the L constructor we use the apo_α^m function to recursively to produce a new sub-structure. When we encounter a R constructor containg a (partially) annotated recursive structure again, we do not go further into recusrion. Point 1 and 2 of the enumeration above show us we are forced to reannotate the top of this structure. Because the coalgebra can possibly create multiple levels of unannotated nodes, we have to use the $fully_{In}$ function. When taken care of the sub-results the result will be wrapped inside an annotation using the ann_{In} function.

Now we can apply the apo_α^m function to our example coalgebra $fromList_{coalg}$ and get back a function that can be used to produce annotated binary trees from a list of integers.

$$\begin{aligned}
&fromList_\alpha^m :: Ann_{In}\ \alpha\ Tree_f\ m \Rightarrow [Int] \rightarrow m\ (Tree_\alpha\ \alpha) \\
&fromList_\alpha^m = apo_\alpha^m\ fromList_{coalg}
\end{aligned}$$

To illustrate the usage of the $fromList_\alpha^m$ function we construct a simple binary tree from a two-element lists. Again we constrain the context to IO and the annotation to $Debug$. The annotation nicely prints out all the sub-structures that are being produced before the final result tree is returned

```
ghci> fromListMA [1, 3] :: IO (FixA Debug (Tree_f Int))
annI: Leaf
```

```
annI: Leaf
annI: Leaf
annI: Branch 3 <D Leaf> <D Leaf>
annI: Branch 1 <D Leaf> <D (Branch 3 <D Leaf> <D Leaf>)>
<D (Branch 1 <D Leaf> <D (Branch 3 <D Leaf> <D Leaf>)>)>
```

Like we did for paramorphisms, we can specialize the apo_α^m function for annotation types that do not require a context to run in. We use the identity monad to get back a pure annotated apomorphism.

$$\begin{aligned}&apo_\alpha :: Ann_{In}\ \alpha\ f\ Identity \Rightarrow \Phi_\alpha\ \alpha\ f\ s \to s \to \mu_\alpha\ \alpha\ f\\&apo_\alpha\ \phi = runIdentity \circ apo_\alpha^m\ \phi\end{aligned}$$

Fixing the annotation to be the identity gives us back a pure apomorphism working over structure without annotations.

$$\begin{aligned}&apo :: Traversable\ f \Rightarrow \Phi_\alpha\ Id\ f\ s \to s \to \mu\ f\\&apo\ \phi = apo_\alpha\ \phi\end{aligned}$$

Now we can simply create a pure *fromList* version working on plain binary trees without annotations.

$$\begin{aligned}&fromList :: [Int] \to Tree\\&fromList = apo\ fromList_{coalg}\end{aligned}$$

In the last two section we have seen how to create recursive destructor and constructor functions using an algebraic approach. The seed and result values for both the paramorphisms and the apomorphisms were polymorph. The next two sections show what happens when we move away from polymorphic result and seed types to using annotated data structures as the result and seed types. This will allow us to write modification functions on our annotated structures, like *insert* and *delete*.

3.3 Endomorphic paramorphism

Both the paramorphisms and the apomorphisms working on annotated structures had enough information to know when to use the ann_{Out} or ann_{In} functions to wrap and unwrap annotations. The paramorphism starts out with querying the value from the annotation before applying the algebra. The apomorphism produces an annotation returned by the coalgebra. The algebras as defined in the previous sections are very general in the sense that they can return a value of any result type *r*. Some paramorphisms might choose to produce a value with a type equal to the input type.

We create two new type synonyms that describe *endomorphic paramorphisms*. The result value is fixed to the same type as the input type $\mu_\alpha\ \alpha\ f$. The second type synonym additionaly hides the annotation variable α inside an existential quantification.

$$\begin{array}{l}\textbf{type}\ \Psi^\varepsilon_\alpha\ \alpha\ f = \Psi_\alpha\ \alpha\ f\ (\mu_\alpha\ \alpha\ f)\\ \textbf{type}\ \Psi^\varepsilon\quad f = \forall\alpha.\Psi^\varepsilon_\alpha\ \alpha\ f\end{array}$$

The Ψ^ε_α type is an specialized version of the Ψ_α type and describes an algbera that returns either an existing fully annotated structure or produces a new fully annotated structure.

$$\begin{array}{l}endo^m_\alpha :: Ann_{IO}\ \alpha\ f\ m \Rightarrow \Psi^\varepsilon_\alpha\ \alpha\ f \to \mu_\alpha\ \alpha\ f \to m\ (\mu_\alpha\ \alpha\ f)\\ endo^m_\alpha\ \psi = ann_{IO}\ (mapM\ fully_{In} \circ out_f \circ \psi \bullet mapM\ (group\ (endo^m_\alpha\ \psi)))\\ \quad \textbf{where}\ group\ f\ c = fmap\ ((,)\ c)\ (f\ c)\end{array}$$

The only real difference between the $para^m_\alpha$ and the $endo^m_\alpha$ function is that the latter knows it needs to use the $fully_{In}$ function on the result of the algbera ψ. The $para^m_\alpha$ function can be used to compute result values of any types from an input structure, even unannotated forms of the input structure. The $endo^m_\alpha$ function can only be used to compute a fully annotated structure with the same type as the input structure.

We illustrate the usage of the endomorphic paramorphisms using the $replicate_{alg}$ function. This algebra describes an operation that distributes a single value to all the value positions in a the binary tree.

$$\begin{array}{ll}replicate_{alg} :: Int \to \Psi^\varepsilon\ Tree_f & \\ replicate_{alg}\ \_\ \ Leaf & = In_f\ Leaf\\ replicate_{alg}\ v\ (Branch\ \_\ (\_, l)\ (\_, r)) & = In_f\ (Branch\ v\ l\ r)\end{array}$$

Because the $replicate_{alg}$ produces a new structure it uses the R constructor from the sum-type. The $endo^m_\alpha$ morphism now knows it should provide new annotations to the top of the result structure using the $fully_{In}$ function.

Combinging the endomorphic paramorphism with the algbera for replication gives us back a true replicate function for annotated structures.

$$\begin{array}{l}replicate^m_\alpha :: Ann_{IO}\ \alpha\ Tree_f\ m \Rightarrow Int \to \mu_\alpha\ \alpha\ Tree_f \to m\ (\mu_\alpha\ \alpha\ Tree_f)\\ replicate^m_\alpha\ v = endo^m_\alpha\ (replicate_{alg}\ v)\end{array}$$

We can now test this function on the result of our prevouis example, the expression $fromList^m_\alpha\ [1,3]$. The result shows a debug trace of how the $replicate^m_\alpha$ function traverses the binary tree and builds up a new tree again with the replicated value.

```
ghci> replicateMA 4 it
annO: Branch 1 <D Leaf> <D (Branch 3 <D Leaf> <D Leaf>)>
annO: Leaf
annI: Leaf
annO: Branch 3 <D Leaf> <D Leaf>
annO: Leaf
annI: Leaf
annO: Leaf
annI: Leaf
annI: Branch 4 <D Leaf> <D Leaf>
annI: Branch 4 <D Leaf> <D (Branch 4 <D Leaf> <D Leaf>)>
<D (Branch 4 <D Leaf> <D (Branch 4 <D Leaf> <D Leaf>)>)>
```

Like for regular paramorphisms we can create a specialized version that works for annotation types that do not require a context to run in. We use the identity monad to get back a pure annotated endomorphic paramorphism.

$$\begin{aligned}&endo_\alpha :: Ann_{IO}\ \alpha\ f\ Identity \Rightarrow \Psi^{\varepsilon}_{\alpha}\ \alpha\ f \rightarrow \mu_\alpha\ \alpha\ f \rightarrow \mu_\alpha\ \alpha\ f\\ &endo_\alpha\ \psi = runIdentity \circ endo^m_\alpha\ \psi\end{aligned}$$

Fixing the annotation to be the identity gives us back a pure endomorphic paramorphism working over structure without annotations.

$$\begin{aligned}&endo :: Traversable\ f \Rightarrow \Psi^{\varepsilon}_{\alpha}\ Id\ f \rightarrow \mu\ f \rightarrow \mu\ f\\ &endo\ \psi = endo_\alpha\ \psi\end{aligned}$$

So, in this section we have seen how to specialize paramorphisms to work with results values of the same type as the input values. Because the result values when working with endomorphic paramorphisms is also an annotated structure, we had to specialize the traversal function to produce annotations for the result structure. In the next section we show how to specialize apomorphisms for have annotated structures as the input seeds.

3.4 Endomorphic apomorphisms

Similar to the concept of endomorphic paramorphisms are the endomorphic apomorphisms. Endomorphic apomorphisms are specific apomorphisms in the sense that the input seed to produce new structures from is itself of the same structure type. Endomorphic apomorphisms working on annotated structures suffer from the same problem as their paramorphic counterparts: the apomorphic traversal function apo^m_α it to generic to reason about annotated seed values.

To allow writing proper endomorphic coalgebras for annotated structure we introduce a two endomorphic coalgbera types.

$$\textbf{type } \Phi^{\varepsilon}_{\alpha}\ \alpha\ f = \Phi_{\alpha}\ \alpha\ f\ (\mu_{\alpha}\ \alpha\ f)$$
$$\textbf{type } \Phi^{\varepsilon}\ f = \forall \alpha . \Phi^{\varepsilon}_{\alpha}\ \alpha\ f$$

The $\Phi^{\varepsilon}_{\alpha}$ type signature fixes the input seed to the type of the structure that will be produced. The additional type Φ^{ε} hides the annotation variable inside an existential quantification.

We will now write the $coendo^{m}_{\alpha}$ function, that takes a endomorphic coalgbera and an fully annotated input structure and produces an fully annotated output structure. Note that both the endomorphic paramorphism and the endomorphic apomorphism modify an input structure to an output structure of the same type. Where the endomorphic paramorphisms takes in input algberas and the endomorphic apomorphisms take coalgberas.

$$\begin{aligned} &coendo^{m}_{\alpha} :: (Traversable\ f, Ann_{IO}\ \alpha\ f\ m) \\ &\qquad \Rightarrow \Phi^{\varepsilon}_{\alpha}\ \alpha\ f \rightarrow \mu_{\alpha}\ \alpha\ f \rightarrow m\ (\mu_{\alpha}\ \alpha\ f) \\ &coendo^{m}_{\alpha}\ \phi = ann_{IO}\ (mapM\ (coendo^{m}_{\alpha}\ \phi\ \text{`}either\text{'}\ fully_{In}) \circ \phi \circ In_{f}) \end{aligned}$$

The $coendo^{m}_{\alpha}$ morphism applies the coalgbera ϕ to the annotated input structure throught the use of the ann_{IO} function from the Ann_{IO} type class. The ann_{IO} function makes sure the input structure is queried from the root annotation and will be supplied a new annotation after applying the specified function. After applying the coalgbera ϕ a case analysis will be done on the result. Either a new seed is produced or an existing structure is reused, similar to the regular apomorphisms. A new seed triggers a new recursive step, an existing structure will be fully annotated.

As an example we will create an endomorphic coalgbera that describes the insertion of one value into a binary tree. The seed value for this coalgbera is an annotated binary tree of which the top level node is guaranteed not to have an annotation[1]. When the input is a *Leaf* we produce a singleton binary tree with the specified input value. Because we use the *R* constructor in the recursive positions of the result, the traversals stops. When the input value is a *Branch* we compare the value to be inserted with the value inside this *Branch*. Based on the result we decide whether to insert the value in the left or right sub-tree. When we want to insert the value into the right sub-tree we stop the recursion with the *R* constructor for the left sub-tree and continue with a new seed using the *L* constructor for the right sub-tree. When we want to insert the value into the left sub-tree we perform the opposite task.

$$\begin{aligned} &insert_{coalg} :: Int \rightarrow \Phi^{\varepsilon}\ Tree_{f} \\ &insert_{coalg}\ v\ (In_{f}\ s) = \end{aligned}$$

[1]The fact that this node is always wrapped inside an In_f constructor and never in an In_α constructor follows from the implementation of the $coendo^{m}_{\alpha}$, but is not encoded in the type. Unfortunately, we cannot change the seed type to $f\ (\mu_\alpha\ \alpha\ f)$ to encode this invariant in the type, because this will also change the type of the seed we have to produce.

```
case s of
  Leaf       → Branch v (R (In_f Leaf)) (R (In_f Leaf))
  Branch w l r →
    case v ‘compare‘ w of
      LT → Branch w (R l) (L r)
      _  → Branch w (L l) (R r)
```

Combining the endomorphic apomorphism with the endomorphic coalgbera for binary tree insertion gives us back a true *insert* function on annotated binary trees.

$$\begin{aligned}&insert^{m}_{\alpha} :: Ann_{IO}\ \alpha\ Tree_f\ m \Rightarrow Int \rightarrow \mu_{\alpha}\ \alpha\ Tree_f \rightarrow m\ (Tree_{\alpha}\ \alpha)\\ &insert^{m}_{\alpha}\ v = coendo^{m}_{\alpha}\ (insert_{coalg}\ v)\end{aligned}$$

We can test the $insert^{m}_{\alpha}$ function by inserting the value 0 into the example tree binary tree produced before with the *fromList* [1,2]. The debug trace shows the traversal that is being performed while inserting a new element into the binary tree.

```
ghci> insertMA 0 it
annO: Branch 1 <D Leaf> <D (Branch 3 <D Leaf> <D Leaf>)>
annO: Leaf
annI: Leaf
annI: Leaf
annI: Branch 0 <D Leaf> <D Leaf>
annI: Branch 1 <D (Branch 0 <D Leaf> <D Leaf>)>
               <D (Branch 3 <D Leaf> <D Leaf>)>
<D (Branch 1 <D (Branch 0 <D Leaf> <D Leaf>)>
             <D (Branch 3 <D Leaf> <D Leaf>)>)>
```

And additionally, in the line of the other morphisms, we define two specializations of this endomorphic apomorphism. The $coendo_{\alpha}$ works for annotations not requiring any context, the *coendo* working on pure structures not requiring any context or annotation.

$$\begin{aligned}&coendo_{\alpha} :: Ann_{IO}\ \alpha\ f\ Identity \Rightarrow \Phi^{\varepsilon}_{\alpha}\ \alpha\ f \rightarrow \mu_{\alpha}\ \alpha\ f \rightarrow \mu_{\alpha}\ \alpha\ f\\ &coendo_{\alpha}\ \phi = runIdentity \circ coendo^{m}_{\alpha}\ \phi\end{aligned}$$

$$\begin{aligned}&coendo :: Traversable\ f \Rightarrow \Phi^{\varepsilon}_{\alpha}\ Id\ f \rightarrow \mu\ f \rightarrow \mu\ f\\ &coendo\ \phi = coendo_{\alpha}\ \phi\end{aligned}$$

So, in this section we have seen how to specialize apomorphisms to work with seed values of the same type as the output values. Because the seed values are now also annotated structures we had to specialize the traversal function to query the structure out of the annotation before using it as the seed.

3.5 Applicative paramorphisms

In section 3.1 we have seen how to write some simple paramorphic algebras. Writing more complicated algebras can be quite hard, mainly because it is not always easy to compose multiple aspects of an operation into one algebra. Combining multilpe algebras into a single algebra that can be used in a single traversal is a well known problem in the world of functional programming. Attribute grammar systems like the *UUAGC*[SPS98] can be used to combine different algebraic operations into a single traversal in an aspect-oriented way. This section describes a more lightweight and idiomatic approach to algebra composition. We make the paramorphic algebra type an instance of Haskell's *Applicative* type class.

First we change the type for the algebra Ψ_α from a type synonym into a real datatype. We change the algebra into a real datatype, because this allows us to add an additional constructor. As we will later see, this extra constructor is needed for the *Applicative* instance. The new Ψ_α datatype will be a generalized algebraic datatypes, or GADT, because we need an existentially quantified type parameter later on.

```
data $\Psi_\alpha$ $(\alpha :: (\star \to \star) \to \star \to \star)$ $(f :: \star \to \star)$ $(r :: \star)$ where
  $Alg :: (f\ (\mu_\alpha\ \alpha\ f, r) \to r) \to \Psi_\alpha\ \alpha\ f\ r$
```

The *Applicative* type class requires us to have a *Functor* instance and requires us to implement two functions, *pure* and $\circledast$.

```
class Functor f ⇒ Applicative f where
  $pure :: \alpha \to f\ \alpha$
  $(\circledast)\ :: f\ (\alpha \to \beta) \to f\ \alpha \to f\ \beta$
```

When we unify the type f with the the Ψ_α type we get the following two functions we have to implement in otder to have an *Applicative* instance. To not get confused with our annotation variable we change the type variable α and β to r and s.

```
$pure :: r \to \Psi_\alpha\ \alpha\ f\ r$
$(\circledast)\ :: \Psi_\alpha\ \alpha\ f\ (r \to s) \to \Psi_\alpha\ c\ f\ r \to \Psi_\alpha\ c\ f\ s$
```

From the signature for $(\circledast)$ it is clear that we should use an algebra that produces a function from $\alpha \to \beta$ and an algebra that produces a α to build an algebra that produces only a β. This applicative sequencing function throws away any information about α. Because both the function and the argument are needed *in every stage of the traversal* we first create a function

$\boxplus$ that simply groups together both the function $r \to s$, the argument r and the result s.

```
(⊞) :: Functor f ⇒ Ψα α f (r → s) → Ψα α f r → Ψα α f (r → s, r, s)
Alg f ⊞ Alg g = Alg $ λx → mk (f (fmap₂ fst₃ x))
                              (g (fmap₂ snd₃ x))
  where mk x y = (x, y, x y)
        fmap₂  = fmap ∘ fmap
```

This grouping function simply collect the results of the two input algebras, the function and the argument, and results in a new algebra that combines these two values and the application of the two.

The next step in our attempt to create an *Applicative* instance for our paramorphic algebras is to add an additional constructor to the Ψ_α datatype. This constructor projects the last component from the triple algebra created by the grouping function. It uses the Ψ_α GADT to create an existential that hides the type variable r inside the *Prj* constructor.

```
Prj :: Ψα α f (r → s, r, s) → Ψα α f s
```

The new constructor requires us the to extend the grouping function with the three additional cases involving the projection constructor. The implementation is quite straightforward, but note that it uses the *pure* function from the *Applicative* instance, which we have not defined yet.

```
(⊞) :: Functor f ⇒ Ψα α f (r → s) → Ψα α f r → Ψα α f (r → s, r, s)
...
Prj f ⊞ Prj g = fmap trd₃ f ⊞ fmap trd₃ g
Alg f ⊞ Prj g = Prj (pure id ⊞ Alg f) ⊞ Prj g
Prj f ⊞ Alg g = Prj f ⊞ Prj (pure id ⊞ Alg g)
```

With the both the grouping function and the projection function we have enough tools to create the *Applicative* instance for paramorphic algebras. The applicative sequencing is a matter of grouping two input algebras together and then only throwing the input types away using the existential from the GADT. The pure function is an algebra that ignores the input structure and always outputs the same value.

```
instance Functor f ⇒ Applicative (Ψα α f) where
  pure  = Alg ∘ const
  α ⊛ β = Prj (α ⊞ β)
```

We require a *Functor* super class instance for this *Applicative* instance which can easily be made in terms of the *Applicative* type class.

```
instance Functor f ⇒ Functor (Ψ_α α f) where
  fmap f h = pure f ⊛ h
```

The *Applicative* instance allows us to easily compose multiple algebras into one, this grouped operation can now be applied to an input structure in one traversal. This composability can help us to create more complicated algebras without creating very big tuples representing all the components at once.

Because the algebra type Ψ_α has become more complicated the $para^m_\alpha$ should be adapted to be able to produce both *Alg* and *Prj* constructors.

```
para^m_α :: Ann_Out α f m ⇒ Ψ_α α f r → μ_α α f → m r
para^m_α (Prj  ψ) = fmap trd_3 ∘ para^m_α ψ
para^m_α (Alg ψ) = return ∘ ψ • mapM (g (para^m_α (Alg ψ))) • ann_Out
  where g f c = fmap ((,) c) (f c)
```

The implementation of this projection aware $para^m_\alpha$ is not very different from our original $para^m_\alpha$. The only difference is that this new version unpacks the projection and applies the inner algebra. After applying only the result value, the third component of the grouped triple, will be returned. The endomorphic variant $endo^m_\alpha$ can be extended in the same trivial way, so we will not show the implementation here.

We can now illustrate the applicative paramorphisms using a well known example, the *repmin* problem. The *repmin* problem describes a single traversal in which every value in a structure, for example our binary tree, will be replaced with the minimum value already in the structure. Our *Applicative* algebra instance allows us to write the two aspects of this operation separately and join them together using the sequencing operator (⊛).

First we write down the algebra for computing the minimum value stored inside a binary tree.

```
min_alg :: Ψ_α Tree_f Int
min_alg = Alg $ λα →
  case α of
    Leaf         → maxBound
    Branch v l r → minimum [v, snd l, snd r]
```

And now we write the replication algebra that produces a *function*. This function produces the new binary tree given some input value.

```
rep_alg :: Ψ_α Tree_f (Int → Tree_α α)
rep_alg = Alg $ λα x →
  case α of
```

$$\begin{aligned}&\mathit{Leaf} && \to \mathit{In}_f\ \mathit{Leaf}\\ &\mathit{Branch}\ \_\ l\ r && \to \mathit{In}_f\ (\mathit{Branch}\ x\ (\mathit{snd}\ l\ x)\ (\mathit{snd}\ r\ x))\end{aligned}$$

Now we can write the *repmin* function by using the *endoMApp* function on the applicative composition of the rep_{alg} and the min_{alg}.

$$\begin{aligned}&\mathit{repmin}^m_\alpha :: \mathit{Ann}_{IO}\ \alpha\ \mathit{Tree}_f\ m \Rightarrow \mathit{Tree}_\alpha\ \alpha \to m\ (\mathit{Tree}_\alpha\ \alpha)\\ &\mathit{repmin}^m_\alpha = \mathit{endoMApp}\ (\mathit{rep}_{alg} \circledast \mathit{min}_{alg})\end{aligned}$$

In this chapter we have seen how to combine multiple algebras into one using the Haskell *Applicative* type class. Writing more complicated operations as a composition of separate algebras is generally more easy than writing monolithic algebras containing multiple aspects in one.

3.6 Lazy IO and strict paramorphisms

The paramorphism function working on annotated structures is as strict as the context associated with the annotation. For example, the debug annotation works in the *IO* monad, which makes all computations strict. This strictness can have severe implications on the running time of the algorithms. In a strict context all the sub results of the recursive computation will be computed, even when the algebra discards them immediately. This is the reason that the debug annotation in example 3.1 prints out far more sub-trees than one would expect from a lazy traversal.

Some monads are lazy by default like the *Identity* monad and the *Reader* monad. Using these lazy monads as the annotation context would make the traversal naturally lazy. Some other monads are strict by default requiring all computations on the left hand side of the monadic bind to be evaluated strictly before the right hand side can be evaluated. Example of these monads are the *IO* monad and the *State* monad. The first thing this section will show is how we can make paramorphisms more lazy on the inside. Only sub-structures should be traversed when the algebra requires them for the computation of the result value. The goal is to make the running time of the paramorphic traversals in strict contexts equivalent to the running time of pure traversals without any context.

The decision about what sub-structures are needed for a certain computation is up to the algebra and not to the paramorphism function itself. The algebra is a pure description that is unaware of the annotation or associated context. Because the paramorphism function does not know what information will be used by the algebra it has to pass in all the recursive sub-results. To clarify this we can look at the $\mathit{contains}_{alg}$ for binary trees.

As input this algebra get a single structure with two booleans in the sub-structures. These booleans are the indication whether the value is contained in one of the sub-structures. Because we are dealing with a lazy language these sub-results are ideally not computed yet, and will only be used when the algebras desires so. Running the paramorphic traversal inside a strict context, like *IO* for our debug annotation, will actually strictly precompute the recursive results before passing them into the algebra. This changes the running time for the $contains^m_\alpha$ function from the expected $O\ (log\ n)$ to an unacceptable $O\ (n)$.

To solve the problem described above we introduce a type class *Lazy* that will allow us to explicitly turn strict monads into lazy ones when this is possible. The only class method is the function *lazy* that gets a monadic computation and turns it into an lazy variant of this computation. Of course this will not be possible in the most general case for all monads.

```
class Monad m ⇒ Lazy m where
  lazy :: m α → m α
```

Both the *Identity* monad and the *Reader* monad are lazy by default and can trivially be made an instance of this type class. To be a bit more general we make the *ReaderT* monad transformer an instance of the *Lazy* class for all cases that it transforms another lazy monad.

```
instance Lazy Identity where
  lazy = id
instance Lazy m ⇒ Lazy (ReaderT r m) where
  lazy c = ask ↣ lift ∘ lazy ∘ runReaderT c
```

Most interesting of all, we can also make *IO* an instance of the *Lazy* class by using the *unsafeInterleaveIO* function. This function takes an *IO* computation and produces an *IO* computation that will only be performed when the result value is needed. This breaks the strict semantics of the *IO* monad, which can become useful for our case.

```
instance Lazy IO where
  lazy = unsafeInterleaveIO
```

Now we have a way to enforce lazy semantics for some of the contexts our traversals may run. By explicitly putting a call to *lazy* just before the recursive invocations in the paramorphism we can make the entire traversal lazy. The laziness of the computation is reflected in the function's class

$$lazyPara^m_\alpha :: (Lazy\ m, Ann_{Out}\ \alpha\ f\ m) \Rightarrow \Psi_\alpha\ \alpha\ f\ r \to \mu_\alpha\ \alpha\ f \to m\ r$$
$$lazyPara^m_\alpha\ \psi = return \circ \psi \bullet mapM\ (group\ (lazy \circ lazyPara^m_\alpha\ \psi)) \bullet ann_{Out}$$
$$\textbf{where}\ group\ f\ c = fmap\ ((,)\ c)\ (f\ c)$$

When we now express the $contains^m_\alpha$ in terms of the more lazy paramorphism.

$$contains^m_\alpha :: (Lazy\ m, Ann_{Out}\ \alpha\ Tree_f\ m) \Rightarrow Int \to \mu_\alpha\ \alpha\ Tree_f \to m\ Bool$$
$$contains^m_\alpha\ v = lazyPara^m_\alpha\ (contains_{alg}\ v)$$

When we now run the $contains^m_\alpha$ functions on an example tree, in this case created by *fromList* [2, 1, 3], we see that the debug trace only prints out the first element of the binary tree, because no other sub-results are needed to come up with the answer.

```
ghci> let tree = fromList [2, 1, 3]
ghci> containsMA 2 tree
ann0: Branch 2  <D Branch 1 <D Leaf> <D Leaf>>
                <D Branch 3 <D Leaf> <D Leaf>>
True
```

The GHCi debugger by default prints out the value the user requests at the prompt, this is the only reason the expression $contains^m_\alpha$ 2 *tree* is even evaluated at all. The traversal has become become lazy on the inside, no unneeded traversals will be performed, but also lazy on the outside, nothing at all will happen until the answer is forced. This behaviour can be compared to the *hGetContents* function from the Haskell prelude which uses lazy *IO* [2] to give back the entire contents of a file or socket resource as a single lazy *String*. This means the actual *IO* actions to read the contents from file will only be performed when the individual characters of the string are inspected. This behaviour can have a strange outcome in practice, evaluation of pure code can have real world side effects. The same is the case for our lazy traversals, the debug annotations only kicks in when the result is inspected. While this behaviour is probably safe for the debug annotation this might not be the case in general. There is a really good reason the *unsafeInterleaveIO* function starts with the word 'unsafe', there are a lot of problems associated with lazy IO[Hu08].

To make sure all the potential side effect stay within the context they belong to we have to make sure our paramorphism remains strict on the outside. We do this by forcing the entire result of the evaluation before returning the answer. Using generic programming we have implemented a generic deep *seq* function that can be used to force an entire computation fully and recursively the moment it would normally only get forced *weak head normal form* (WHNF). The implementation internally uses the *seq* function from the Haskell prelude that forces the first argument WHNF when the second argument gets forced.

[2]The *hGetContents* function internally also uses the *unsafeInterleaveIO* function.

$$\begin{aligned}&gdseq :: \alpha \to \alpha\\&seq \quad\;\; :: \alpha \to \beta \to \beta\end{aligned}$$

We now apply the *gdseq* to the result of the paramorphism.

$$\begin{aligned}&para^{m}_{\alpha} :: (Lazy\ m, Ann_{Out}\ \alpha\ f\ m) \Rightarrow \Psi_{\alpha}\ \alpha\ f\ r \to \mu_{\alpha}\ \alpha\ f \to m\ r\\&para^{m}_{\alpha}\ \psi = gdseq\ \text{`}liftM\text{`}\ lazyPara^{m}_{\alpha}\ \psi\end{aligned}$$

By creating a new paramorphism function that forces the result before returning we get a traversal that is lazy internally but is strict on the outside. As we will see later when dealing with structures annotated with the persistent annotations, this evaluation semantics is essential.

Chapter 4

Persistent Heap

The previous chapter explained how to use generic programming with annotated fixed points to associate custom functionality to traversals over recursive datatypes. The goal of this project is to use this annotation framework to project operations on recursive datatypes to a persistent instance on the computer's hard disk. To make this goal possible we are in need for a flexible storage structure that allows us to freely read and write data to disk.

In this chapter we introduce a storage heap that uses generic binary serialization for values of arbitrary Haskell datatypes on disk. The structure of this heap is very similar to the structure of in-memory heaps used by most programming language implementations to store pointer data. The heap allows the basic operations of allocating new data blocks, freeing existing data blocks, reading from and writing to data blocks. The current implementation of the heap structure that ships with this project is not fully optimized for speed and is just a prototype implementation. One single heap structure is stored in one single file that can shrink and grow in size on demand.

All operations on the heap work in a monadic context that are built on top of the *IO* monad. To make it easier to reason about the behaviour of the storage heap we introduce a stacked heap monad. We define one context for *read-only* access to the underlying data, one context for *read-write* access to the data, and one context for *allocating and freeing* new blocks of data. We call the read-only context $Heap_R$, the read-write context $Heap_W$ and the allocation context $Heap_A$. The top context is the $Heap_W$, which internally uses the $Heap_A$ context, which on its turn uses the $Heap_R$ on the inside. This separation of context is introduced to be able to differentiate between the possible types of actions. As we will see in the next chapter **todo:** xxx this separation is essential for gaining control over the laziness of our storage framework.

In this chapter we introduce zeven basic heap operations, some of which we will be using in the next chapters, some of which will only be used internally.

$$
\begin{array}{lll}
\mathit{allocate} & :: \mathit{Integer} & \rightarrow \mathit{Heap}_A\ (\mathit{Pointer}\ \alpha) \\
\mathit{free} & :: \mathit{Pointer}\ \alpha & \rightarrow \mathit{Heap}_A\ () \\
\mathit{update} & :: \mathit{Binary}\ \alpha \Rightarrow \mathit{Pointer}\ \alpha \rightarrow \mathit{ByteString} & \rightarrow \mathit{Heap}_W\ () \\
\mathit{read} & :: \mathit{Binary}\ \alpha \Rightarrow \mathit{Pointer}\ \alpha & \rightarrow \mathit{Heap}_R\ \alpha \\
\mathit{fetch} & :: \mathit{Binary}\ \alpha \Rightarrow \mathit{Pointer}\ \alpha & \rightarrow \mathit{Heap}_W\ \alpha \\
\mathit{write} & :: \mathit{Binary}\ \alpha \Rightarrow \alpha & \rightarrow \mathit{Heap}_W\ (\mathit{Pointer}\ \alpha) \\
\mathit{run} & :: \mathit{FilePath} & \rightarrow \mathit{Heap}_W\ \alpha \rightarrow \mathit{IO}\ \alpha
\end{array}
$$

The first three operations are potentially unsafe and will only be used as building blocks for the four last functions. We will explain these heap operations after showing the global layout of the heap file.

4.1 Heap layout

On disk the heap consists of one linear contiguous list of blocks. A block is a single atomic storage unit that can contains an arbitrarily large string of bytes. Every block consist out of a single byte indicating whether the block is occupied with data or free for allocation, a four byte size indication representing the byte size of the payload and the payload itself which is a stream of bytes. The size of the payload is always equal to the byte size stored in the block header.

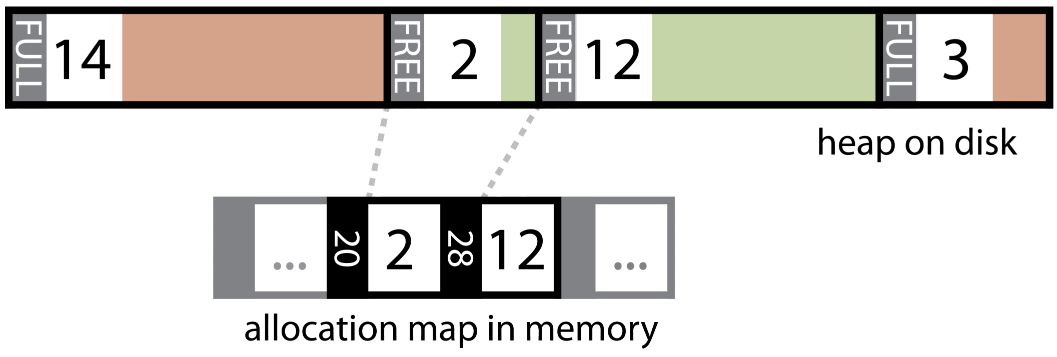

Figure 4.1: Storage heap with 2 occupied and 2 free blocks.

The Haskell code uses a *Pointer* datatype to refer to these storage blocks. Pointers are just integer values pointing to the exact byte offset of the block in the file that stores the heap. The *Pointer* datatype is indexed with a phantom type indicating the original type of the data stored in the block. Because *Pointer*s are stored on disk we need a *Binary* instance for it, which can be derived automatically because the type is just a **newtype** over an *Integer*.

```
newtype Pointer α = Ptr {unPtr :: Integer}
  deriving Binary
```

Before we continue with the description of the heap operations we quickly explain the *Binary* type class. The class contains two methods, *get* and *put*.

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

The *get* and *put* methods can be used to build computations for deserializing and serializing values from and to *ByteString*s. Take this example *Binary* instance for the *Either* α β type, defined in the `Data.Binary` package.

```
instance (Binary α, Binary β) ⇒ Binary (Either α β) where
  put (L α) = do putWord8 0; put α
  put (R β) = do putWord8 1; put β
  get = do w ← getWord8
           case w of
             0 → liftM L get
             _ → liftM R get
```

When an instance for *Binary* is defined for a certain type we can use the *decode* and *encode* functions from the library to convert strings of bytes to Haskell values and vice versa.

```
encode :: Binary α ⇒ α → ByteString
decode :: Binary α ⇒ ByteString → α
```

The following example shows how to use the *encode* and *decode* functions.

```
ghci> encode (Right 'a' :: Either Bool Char )
Chunk "\SOHa" Empty
ghci> decode it :: Either Bool Char
Right 'a'
```

From the example *Binary* instance for the *Either* α β type we can see that *Binary* instances are built very systematically. Using the generic programming library *Regular* we have created two generic functions to automatically derive the *get* and *put* methods for arbitrary Haskell types.[1]

4.2 Reading

We can read from a storage block using the *read* operation. The read operation takes a pointer to a storage block and tries to return a value of the Hakell datatype indicated by the phantom type in the input pointer. When the operation fails, possibly due to an invalid pointer or some I/O error, the function throws an exception. When the functions succeeds to read the stream of bytes from disk it uses *get* function from the *Binary* type class to deserialize the data to a true Haskell value.

The *read* functions runs inside the read-only heap context $Heap_R$. This context is defined as a *ReaderT* monad transformer wrapping *IO*. The value stored inside the reader monad is the file *Handle* of the heap file.

```
newtype Heap_R α = Heap_R (ReaderT Handle IO α)
  deriving (Functor, Applicative, Monad, MonadIO)
```

We now give the implementation of the *read* function in pseudo code.

```
read :: Binary α ⇒ Pointer α → Heap_R α
read ptr =
  do file ← ask                  -- Get the heap file handle.
     liftIO (seek file (offset ptr))-- Move file cursor to pointer offset.
     s  ← readBlockSize          -- Read the block size from the header.
     bs ← readPayload s          -- Read the ByteString from the current payload.
     return (decode bs)          -- Decode the bytes to a Haskell value.
```

The actual code involves additional sanity checks that are left out in this example. The *read* function simply moves the current file position to the start of the block and reads the size of the block from the header. The header size is used to read the payload into a *ByteString*. This *ByteString* is decoded and returned as a true Haskell value.

[1]The code for this package can be found at:
http://hackage.haskell.org/package/regular-extras

4.3 Allocating and freeing

When an application requires to store a new value on the storage heap, we need to allocate a new data block with the right size. The *allocate* functions allocates new blocks by manipulating an in-memory *AllocationMap*. This datatype represents a mapping from block offsets to a list of payload sizes, for all blocks on the storage heap that are currently unoccupied. Additionally, the map stores the total size of the heap.

```
data AllocationMap = AllocationMap
  {map :: IntMap [Int]
  ,size :: Int
  }
```

The *allocate* function takes an integer value indicating the size the new block should at least have. The function returns a *Pointer* to an exact block offset. The block pointed to by this offset can now we be used to store information. The phantom type of the *Pointer* is still polymorphic because the allocation function does not require this information.

```
allocate :: Integer → Heap_A (Pointer α)
```

The *allocate* function performs a lookup in the *AllocationMap* for a block of sufficient size. When there is no large enough free block available the *allocate* function can join up two or more consecutive free block and use this joined result block. When joining two blocks is not possible the *allocate* function creates a new block with sufficient size at the end of the heap, thereby growing the heap. Because all the administration is in-memory, the allocation itself does not require any disk access.

As the type signature of *allocate* indicates there is a separate $Heap_A$ context for the allocation operations. The $Heap_A$ context uses a *ReaderT* monad transformer to supply all operations with the current *AllocationMap*. This contexts internally wraps the $Heap_R$ context.

```
newtype Heap_A α = Heap_A (ReaderT AllocationMap Heap_R α)
  deriving (Functor, Applicative, Monad, MonadIO)
```

Besides allocation of new blocks, the $Heap_A$ context also supports the possibility to free existing blocks.

```
free :: Pointer α → Heap_A ()
```

This operation add the block pointed to by the input *Pointer* to the allocation map. The block can now be reused when needed. This operation also runs in-memory and does not require any disk access.

4.4 Writing

The third heap context is the writing context $Heap_W$ that allows write access to the heap structure. The $Heap_W$ context directly wraps the $Heap_A$ context, thereby allowing computations to write, allocate, free and read.

```
newtype HeapW α = HeapW (HeapA α)
  deriving (Functor, Applicative, Monad, MonadIO)
```

To be able to also perform read and allocation operations in the $Heap_W$ context, we define two lift functions *liftR* and *liftA* that lift operation into the read-write context.

```
liftR :: HeapR α → HeapW α
liftA :: HeapA α → HeapW α
```

The first function that runs inside the $Heap_W$ context is the *update* function. This function takes a value of some Haskell datatype and a pointer to a block and updates the payload of that block with the binary serialization of the value. The function uses the *put* function from the *Binary* type class to perform the serialization.

We show the implementation of the update function in pseudo code.

```
update :: Binary α ⇒ Pointer α → ByteString → HeapW ()
update ptr bs =
  do file ← ask
     liftIO (seek file (offset ptr))-- Move file cursor to pointer offset.
     s ← readBlockSize             -- Read block size from the header.
     when (s < length bs)          -- When payload to small...
       (throw SomeError)           -- ...throw some error.
     writePayload bs               -- Write bytes to the payload.
```

This update function is very useful but unsafe, when the payload of the input block is not large enough to hold the string of bytes this function cannot write the value to disk. When there is not enough data available in the block the function throws an exception.

Because the *update* function is unsafe we implement a safe function *write* on top of it. The *write* operation only takes a value as input and itself allocates a storage block just big enough to hold the serialized value. The payload of the allocated block is updated with the serialized value, which now always succeeds. The function returns a pointer to the allocated block.

```
write :: Binary α ⇒ α → HeapW (Pointer α)
write α =
do let bs = encode α
   block ← liftA (allocate (fromIntegral (Bs.length bs)))
   update block bs
   return block
```

It can be useful to read a block from the heap and then immediately free it, because it is guaranteed not to be read again. For this purpose, we define a function *fetch* that is a combination of a *read* and a *free*.

```
fetch :: Binary α ⇒ Pointer α → Heap_W α
fetch p = do r ← liftR (read p)
             liftA (free p)
             return r
```

We lift this function to the $Heap_W$ context so we can easily run it side by side with the write function, without first lifting it.

4.5 Root node

The storage heap layout does not force any structure on the data one might save. The only requirement is that the storage is block based, but no relations between separate blocks is required. As we will see in chapter 5, we will eventually reuse the structure of recursive datatypes to guide the way we make relations between individual blocks.

Every data structure has at least one root node and for every operation we need this root node. To make the root node accessible, we make the first storage block special and dedicate it to store the root of our data structure. We call this block, at offset 0, the *null block*. We define three helper functions for working with the assumption the null block stores a pointer to the root of the data structure root.

The *query* function asks a read-only heap action that takes as input the root of the data structure. Altough the action will be lifted to the root heap context, the $Heap_W$, the action itself is cannot perform write actions.

```
query :: (Pointer f → Heap_R c) → Heap_W c
query c = liftR (read (Ptr 0) ↣ c)
```

The *produce* function is used to create new structures from scratch. After the producer function has built up the data structure, a pointer to the structure root will be stored in the null block.

```
produce :: Heap_W (Pointer f) → Heap_W ()
produce c = c ↣ update (Ptr 0)
```

The *modify* function takes a computation that transforms an existing structure into a new structure. The original structure is read from the null block, a pointer to the newly created structure will be stored at the null block again.

$$
\begin{aligned}
&\mathit{modify} :: (\mathit{Pointer}\ f \rightarrow \mathit{Heap}_W\ (\mathit{Pointer}\ f)) \rightarrow \mathit{Heap}_W\ ()\\
&\mathit{modify}\ c = \mathit{liftR}\ (\mathit{read}\ (\mathit{Ptr}\ 0)) \rightarrowtail c \rightarrowtail \mathit{update}\ (\mathit{Ptr}\ 0)
\end{aligned}
$$

These functions are rather useless when manually storing blocks of data in the storage heap, but become very useful when the heap is used to store true data structures. We will see this in action in section 5.3.

4.6 Running the heap operations

Now that we have defined three different heap contexts together with some basic operations for reading, writing, allocating and freeing blocks of data we can define a *run* function that can apply these operations to a file.

The *run* function takes a path to the file containing the heap structure and a computation in the read-write context and performs the operations on the file.

$$\mathit{run} :: \mathit{FilePath} \rightarrow \mathit{Heap}_W\ \alpha \rightarrow \mathit{IO}\ \alpha$$

The *run* function only works for the Heap_W context because this is the topmost context. Operations in the Heap_A and Heap_R contexts must be lifted before they can be performed. The *run* function start by opening the file and storing the file handle in the state monad defined by the Heap_R context. It then traverses all blocks on the heap to generate the in-memory allocation map with all the free blocks. The resulting map is stored inside the Heap_A context. After the allocation map is read the *run* function applies the input operation to the heap structure. When the computation has finished the blocks in the heap on file are synchronized with the possibly updated allocation map.

The following example shows how we can open up an existing heap, store two strings, read one string back in and print the result. The block is freed immediately after using it. When the computation is finished the heap on file contains one additional block containing the second string. When the file does not exists an new empty heap is created.

```
main :: IO ()
main = run "test.heap" $
  do p₀ ← write "First block"
```

```
_   ← write "Second block"
str ← liftR (read p0)
liftIO (putStrLn str)
liftA (free p0)
return ()
```

Because the entire heap is stored in the `test.heap` file, we can use invoke this mini-program several times consecutively. Each time the function runs, a new block is left on the heap containing the value `"Second Block"`.

In this section we have described a very basic file based heap structure with a simple interface that allows us to allocate and write new blocks of information, to read from existing blocks of information and to free blocks that are no longer of any use. The heap manages allocation and reuse of blocks of data, but does not know of any structure of the information. Any relation between blocks of data is up to the user of the heap. The next chapter shows how we can connect our generic annotation framework for recursive datatypes to the storage heap to obtain a generic storage framework.

Chapter 5

Generic storage

In chapter 2 and 3 we have built a framework for generic traversals over annotated recursive data structures. In chapter 4 we have shown a disk based a storage heap that can be used to store arbitrary blocks of binary data. In this chapter we will show how to combine our generic annotation framework with the storage heap to derive a generic storage framework.

We will use the *Pointer* type to create an annotation that maps non-recursive nodes to individual storage blocks on the storage heap. Using these annotations we will lift operations working on the recursive datatypes to one of the heap contexts. These lifted operation will now no longer operate in-memory but read there input, and write back their output, to an on-disk representation.

5.1 Pointer annotation

In section 4.1 we have seen the *Pointer* type that stores an offset into the storage heap. This pointer is represented by an integer value. When we want to represent recursive datatypes that live on our storage heap, we can use these pointers as the annotations for our fixed points. Using pointers at the recursive positions slices a recursive data structure into individual non-recursive pieces. Each piece is indirectly connected using a *Pointer* instead of a regular in-memory connection.

The *Pointer* datatype itself cannot be used as an annotation because it has kind $\star \rightarrow \star$, while the annotations in our framework have kind $(\star \rightarrow \star) \rightarrow (\star \rightarrow \star)$. To be able to still use pointers as annotations we define a simple wrapper type $Pointer_\alpha$ with the right kind.

$$\textbf{newtype}\ Pointer_\alpha\ f\ \alpha = Pointer_\alpha\ \{unP :: Pointer\ (f\ \alpha)\}$$

```
    deriving Binary
```

We can now represent a persistent version of our binary tree with the following type.

```
  type PersistentTree = $\mu_\alpha$ Pointer$_\alpha$ Tree$_f$
```

The *PersistentTree* describes a recursive datatype, but the recursion is indirect. All connections between *Branch* nodes to other *Branch* nodes or *Leaf* nodes are described as pointer to locations on the heap. In figure 5.1 we see an example of an binary annotated with the *Pointer*$_\alpha$ annotation.

5.2 Annotation instances

In chapter 3 dealing with generic traversals over recursive datatypes, we have implemented annotation aware paramorphism and apomorphism. These traversal functions can be used to both destruct and construct recursive datatype with annotated fixed points. For the binary tree example we seen some simple algebras and coalgebras to describe the semantics of the traversal. In order to be able to apply these algebras to our persistent binary tree we have to make the *Pointer*$_\alpha$ type an instance of the annotation type classes.

We first make *Pointer*$_\alpha$ an instance of the *Ann*$_{Out}$ type class. We can simply use the *read* function from our read-only storage heap for the implementation. This read-only annotation can be associated with the *Heap*$_R$ context. We first unpack the fixed point and the *Pointer*$_\alpha$ wrapper and than read one non-recursive node from disk. Because the storage heap only stores plain strings of bits a *Binary* instance is needed to deserialize the Haskell value. The *ann*$_{Out}$ function takes a pointer to one non-recursive node, possibly containg pointers to other nodes again, and return this node, with type f (μ_α *Pointer*$_\alpha$ f). When the fixed point does not contains an annotation at all we just unpack the node like we did with the other annotations instances,

```
  instance (Traversable f, Binary (f ($\mu_\alpha$ Pointer$_\alpha$ f)))
     ⇒     Ann$_{Out}$ Pointer$_\alpha$ f Heap$_R$ where
    ann$_{Out}$ (In$_\alpha$ (Pointer$_\alpha$ f)) = read f
    ann$_{Out}$ (In$_f$            f  ) = return f
```

The *Ann*$_{In}$ instance for the *Pointer* type is just a matter of writing a single non-recursive node to disk and storing the pointer inside the *Pointer*$_\alpha$ wrapper type. This action can only be done inside the read-write *Heap*$_W$ context.

```
  instance (Traversable f, Binary (f ($\mu_\alpha$ Pointer$_\alpha$ f)))
     ⇒     Ann$_{In}$ Pointer$_\alpha$ f Heap$_W$ where
```

$$ann_{In} = fmap\ (In_\alpha \circ Pointer_\alpha) \circ write$$

Sometimes, when working inside the read-write heap context $Heap_W$, we also want to be able to perform read-only actions, so we also give an Ann_{Out} instance for the *Pointer* type inside the $Heap_W$ context. The implementation is very similar to the one for the read-only $Heap_R$ context, except we no lift teh *read* operation to the read-write context.

$$\begin{aligned}
&\textbf{instance}\ (Traversable\ f, Binary\ (f\ (\mu_\alpha\ Pointer_\alpha\ f)))\\
&\quad \Rightarrow \quad Ann_{Out}\ Pointer_\alpha\ f\ Heap_W\ \textbf{where}\\
&\quad ann_{Out}\ (In_\alpha\ (Pointer_\alpha\ f)) = liftR\ (read\ f)\\
&\quad ann_{Out}\ (In_f\ \qquad\qquad\ f\ ) = return\ f
\end{aligned}$$

Because we now both have an Ann_{Out} and an Ann_{In} instance for the *Pointer* type inside the $Heap_W$ context we can create an instance for the Ann_{IO} typeclass. The Ann_{IO} instance assumes the value that gets modified will not be needed again in its original form. We will use the *fetch* function to remove the original block from disk.

$$\begin{aligned}
&\textbf{instance}\ (Traversable\ f, Binary\ (f\ (\mu_\alpha\ Pointer_\alpha\ f)))\\
&\quad \Rightarrow \quad Ann_{IO}\ Pointer_\alpha\ f\ Heap_W\ \textbf{where}\\
&\quad ann_{IO}\ f\ (In_\alpha\ (Pointer_\alpha\ h)) = fmap\ (In_\alpha \circ Pointer_\alpha)\ (write \leftarrowtail f \leftarrowtail fetch\ h)\\
&\quad ann_{IO}\ f\ (In_f\ \qquad\qquad\ h\ ) = fmap\ (In_\alpha \circ Pointer_\alpha)\ (write \leftarrowtail f\ h)
\end{aligned}$$

In order to store the recursive structures on disk we also need a *Binary* instance for the annotated fixed point operator type itself. This is a partial instance because we will not allow to store unannotated fixed points. This means the structure should first be fully annotated using the $fully_{In}$ function.

$$\begin{aligned}
&\textbf{instance}\ Binary\ (\alpha\ f\ (\mu_\alpha\ \alpha\ f)) \Rightarrow Binary\ (\mu_\alpha\ \alpha\ f)\ \textbf{where}\\
&\quad put = put \circ out_a\\
&\quad get = liftM\ In_\alpha\ get
\end{aligned}$$

These instances are the only thing we need to connect our generic annotation framework to our storage heap implementation. In the next section we will see how we can use the building blocks described upto now to build and inspect a true binary tree on disk.

5.3 Persistent binary tree

In chapter 3 we have implemented some example algebras and coalgebras working on binary trees. These functions could be lifted to true functions using the $para_{\alpha}^{m}$ and apo_{α}^{m} functions. When lifted the function work for an annotated binary tree in some monadic context. The context has been associated with the annotation type using one of the annotation type classes. Recall these four functions working on binary trees.

$$
\begin{array}{lllll}
fromList_{\alpha}^{m} & :: Ann_{In} & \alpha\ Tree_f\ m \Rightarrow [Int] & & \rightarrow m\ (Tree_{\alpha}\ \alpha) \\
contains_{\alpha}^{m} & :: Ann_{Out} & \alpha\ Tree_f\ m \Rightarrow Int \rightarrow Tree_{\alpha}\ \alpha & & \rightarrow m\ Bool \\
insert_{\alpha}^{m} & :: Ann_{IO} & \alpha\ Tree_f\ m \Rightarrow Int \rightarrow Tree_{\alpha}\ \alpha & & \rightarrow m\ (Tree_{\alpha}\ \alpha) \\
repmin_{\alpha}^{m} & :: Ann_{IO} & \alpha\ Tree_f\ m \Rightarrow Tree_{\alpha}\ \alpha & & \rightarrow m\ (Tree_{\alpha}\ \alpha)
\end{array}
$$

Using the annotation instances for the $Pointer_{\alpha}$ type we can now lift these four function to work on persistent binary trees. The only thing we have to in order to achieve this is specialize the types. The persistent versions of these functions become:

$$
\begin{array}{l}
fromList_P :: [Int] \rightarrow Heap_W\ PersistentTree \\
fromList_P = fromList_{\alpha}^{m} \\
contains_P :: Int \rightarrow PersistentTree \rightarrow Heap_R\ Bool \\
contains_P = contains_{\alpha}^{m} \\
insert_P :: Int \rightarrow PersistentTree \rightarrow Heap_W\ PersistentTree \\
insert_P = insert_{\alpha}^{m} \\
repmin_P :: PersistentTree \rightarrow Heap_W\ PersistentTree \\
repmin_P = repmin_{\alpha}^{m}
\end{array}
$$

The implementations can just be reused because the original functions are generic enough to work for all annotations that have an instance for the Ann_{In}, Ann_{Out} and Ann_{IO} classes. Note that the read-only functions are lifted to the $Heap_R$ context, while the read-write functions are lifted to the $Heap_W$ context.

These persistent operations can now be applied against the storage heap. We define an example function that opens a heap file, builds an initial tree from a list, inserts one item to the list and then replaces all the value with the minimum value of the tree.

$$
\begin{array}{l}
buildTree :: FilePath \rightarrow IO\ () \\
buildTree\ file = run\ file\ \$ \\
\quad produceP\ \$\ fromList_P\ [5,3,2,4,1] \rightarrowtail \\
\qquad\qquad insert_P\ 4 \rightarrowtail \\
\qquad\qquad repmin_P
\end{array}
$$

The *produceP* function is a simple wrapper around *produce* (from section 4.5) that works for the *PersistentTree* type, instead of the *Pointer* type directly.

$$
\begin{array}{l}
produceP :: Heap_W\ (\mu_{\alpha}\ Pointer_{\alpha}\ f) \rightarrow Heap_W\ () \\
produceP\ c = produce\ (liftM\ (unP \circ out_a)\ c)
\end{array}
$$

And, as a second example, we define a function that opens a heap file and checks whether the tree stored on the heap contains the value 1. The boolean result will be printed to the standard output.

```
inspectTree :: FilePath → Int → IO ()
inspectTree file i = run file $
  do j ← queryP (contains_P i)
     liftIO (print j)
```

The *queryP* function is a simple wrapper around *query* (also from section 4.5) that works for the *PersistentTree* type, instead of the *Pointer* type directly.

```
queryP :: (μ_α Pointer_α f → Heap_R c) → Heap_W c
queryP c = query (c ∘ In_α ∘ Pointer_α)
```

Now we can run these operations consecutively and see the expected result: the tree written to disk contains the value 1 (which was the minimum when we applied *repmin*$_P$) and does not contain the value 3.

```
ghci> buildTree "tree.db"
ghci> inspectTree "tree.db" 2
True
ghci> inspectTree "tree.db" 3
False
```

So we have specialized some generic operations working on annotated binary trees to work for the *Pointer*$_\alpha$ annotation in the heap contexts and were able to run the operations with the *run* function from our storage heap. Although the example invocations look very simple and produce the expected result, in the background a lot is going on. Let us try to explain what happens when running these operations.

When we run the command `buildTree "tree.db"` the *run* function will open up a file containing a storage heap and will apply the given *Heap*$_W$ operation to this heap. We will now give a step-by-step overview of the important steps that will be happen internally.

- The *run* function first opens the specified file in read-write mode and starts scanning the file. As explained in section 4.6 all blocks will be traversed to build up an in-memory allocation map. This map can be used for the allocation of new blocks and freeing of existing blocks of binary data.
- After reading the allocation map the *run* function starts the actual heap operation inside the wrapped *produce* function. Our operations

is a monadic sequencing of four operations, so the $Heap_W$ computation naturally starts with the first, the $fromList_P$ [5, 3, 2, 4, 1]. Recall from section 4.4 that the $Heap_W$ context internally wraps both the $Heap_A$ context from section 4.3 and the $Heap_R$ context from section 4.2. This monad transformer stack internally provides the allocation map and heap file handle to all operations.

- The $fromList_P$ function is built up from the $fromList_{coalg}$, which is lifted to a true function using the apomorphic traversal apo^m_α. The definition of apo^m_α in section 3.2 shows it will use the coalgebra to corecursively create a new structure. At every step in the recursive generation of the structure the individually created nodes will be wrapped inside an annotation using the ann_{In} function.

  In our case the annotation type is specialized to the $Pointer_\alpha$ type. This means that for every node that the $fromList_{coalg}$ produces, the apomorphisms will use the ann_{In} function for the $Pointer_\alpha$ instance to wrap the node inside the $Pointer_\alpha$ annotation.

  Now recall the definition of the $AnnIn$ instance for $Pointer_\alpha$.

  $$ann_{In} = fmap\ (In_\alpha \circ Pointer_\alpha) \circ write$$

  This definition means that wrapping a node inside the $Pointer_\alpha$ annotation means writing the binary representation of the node to disk in a freshly allocated block and storing the offset to that block in side the $Pointer_\alpha$ constructor. For example, the invocation of $ann_{In}\ Leaf$ will result in something like $In_\alpha\ (Pointer_\alpha\ (Ptr\ 234))$. This implies that a value of type $\mu_\alpha\ Pointer_\alpha\ Tree_f$ only contains a wrapped offset to a place on disk where the actual node can be found.

  So, when the $fromList_P$ function is finished all the produced *Branch* and *Leaf* nodes will be stored in binary form, each on their own block, in the storage heap. All *Branch* nodes will have pointers at the recursive positions to other heap blocks containing the sub-tree. Figure 5.1 shows an example of a persistent binary tree. The result variable p now contains a pointer to the root of the binary tree.

- The next operation is the $insert_P\ 4\ p$, which insert the value 4 in the binary tree created by the previous operation. This operation works slightly different from the $fromList_P$ operation, because it works on an existing binary tree. The $insert_P$ functions is built up from the $insert_{coalg}$ and lifted to a true function using the endomorphic apomorphism $coendo^m_\alpha$.

  The implementation of $coendo^m_\alpha$ (see section 3.4) differs from the implementation of apo^m_α, the $coendo^m_\alpha$ traversal uses an existing binary tree as the input seed. This difference becomes clear from the definition, the function uses the ann_{IO} modifier function.

So the traversal gets as input an annotated structure, in our case a μ_α $Pointer_\alpha$ $Tree_f$, and uses the pointer to read the actual node from disk. This node will be passed to the endomorphic coalgebra which produces either a new seed or a new sub structure. When the coalgebra finishes the Ann_{IO} will make sure the result is stored to disk again. The usage of the Ann_{IO} function forces every node that gets touched to be fetched (and also removed) from and saved to disk[1]. This behaviour is very similar to what happens in regular in-memory update functions: inserting a node into a binary tree requires a *copy* of the entire path up to the place where the new nodes gets inserted. All sub-trees that are not needed for the inserting will not be touched. This makes the asymptotic running time of the persistent algorithms equal to that of the regular in-memory variants.

So, when the $insert_P$ function finishes it has inserted a new node into the existing tree on disk. When doing this the entire path to this node has been updated including the root node. All old nodes are removed and the freed blocks can be reused. The new root node is returned and represents the new binary tree.

- The next step is the $repmin_P$ function, which is also a modifier function. The $repmin_P$ function uses more or less the same techniques as the $insert_P$ function, however $repmin_P$ is a combination of algebra (min_{alg}) and an endomorphic algebra (rep_{alg}). After running the $repmin_P$ algorithm the tree on disk will no only contain six times the number 1.

- When the previous operations are finished the root pointer of the final binary tree is saved in the variable r. This variable is stored as the root of the structure inside the null block by the surrounding *produce* function section 4.5.

  Now the run function terminates and the heap file will be closed. The heap will contains a sliced binary serialization of a binary tree containing 6 times the value 1.

- Now we run the second example *inspectTree*, which performs a read-only operation inside the *query* function. The *query* function (from section 4.5) read the root of the data structure from the null block and supplies this to the specified operation. The $contains_P$ function is used to check for the existence of a value 1 inside the persistent binary tree. The paramorphic operation performs a traversal over the persistent

[1]See section **todo:** TODO for a possible optimization.

binary tree and soon figures out the value 1 is indeed stored inside the tree. Assuming the $contains_{alg}$ here is lifted using the paramorphism resulting from section 3.6 the traversal will be lazy.

Both the *buildTree* and the *inspectTree* are two distinct functions that individually open up the heap file and perform their operations. This means that both function can be compiled to different programs that run consecutively without any shared state except the storage heap.

5.4 Overview

In this chapter we have seen how to connect the low level interface of our storage heap to our generic annotation framework. By creating annotation type class instances for the *PointerA* annotation, we were able to derive a generic storage framework. All generic operations, in our example for binary trees, that are specialized to work for the *PointerA* annotation will out of the box work for persistent data structures.

The original algebraic operations for our binary trees are annotation-unaware, only when they are lifted and specialized they can be used to manage data structures stored on disk.

Generically annotating algebraically defined operations to work in a persistent computational context is the main achievement of this project. This achievement makes it possible to write purely functional algorithms on purely function data structures, but still be able to use them in a non-pure environment.

The framework defined here allows for a wide range of extensions, most of which are not yet implemented or even described in this report. In chapter 8 we will describe some limitation of this system and possible future extensions.

The current framework is generic because it works for all regular recursive datatypes. Most, but not all, common functional data structures are regular recursive datatypes. In the next chapter we will show some example of recursive data structures that cannot be used in our current framework. The next chapter will show what is needed to extend the framework to also be usable for indexed datatypes like GADTs.

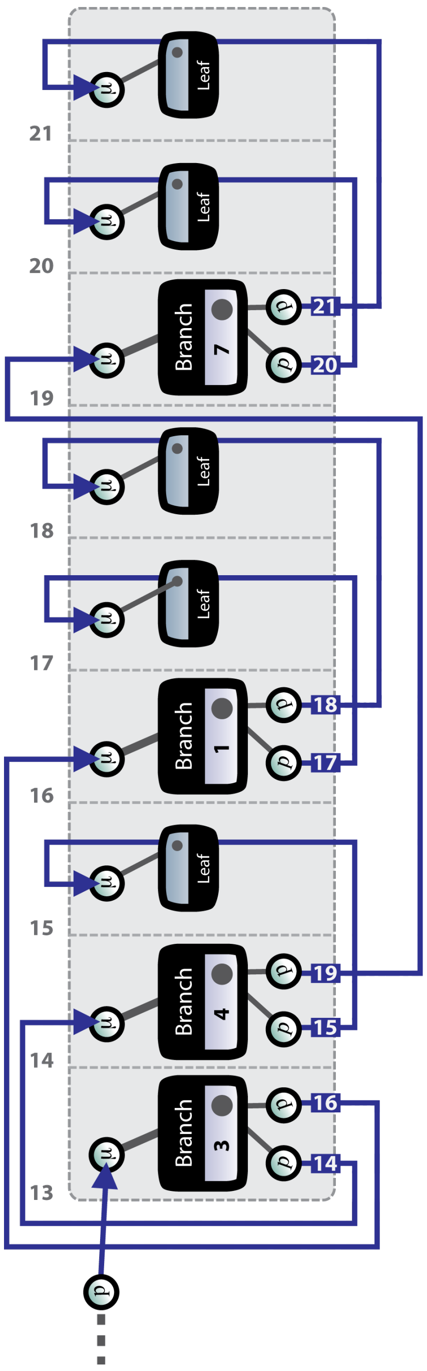

Figure 5.1: This image shows the same binary tree as in figure 2.1 and figure 2.2, but this time annotated with the *Pointer* annotation and laid out on the storage heap. Every node is stored in its own heap block and at a certain offset. Every *Branch* nodes refers to other node using the pointer annotation that stores the integer offset of the heap block the target can be found in.

Chapter 6

Indexed Datatypes

In the previous chapters we have shown how to build a generic storage framework for recursive data structures. This framework only works for regular datatypes, types in which the recursive positions can only refer to the exact same type again. The system will not work for any-non regular datatypes like mutually recursive datatypes[RHLJ08], nested datatypes[BM98] and indexed datatypes like generalized algebraic datatypes or GADTs[Joh08]. In this section we will show how to extend our current system to work with non-regular datatypes. First we will explore some examples of non-regular datatypes.

- *Mutually recursive datatypes*, or *indirect recursive datatypes*, are types in which the recursive positions refer to other datatypes that directly or indirectly refer back to the original. Rodriguez et al.[RHLJ08] show an interesting example of a mutual recursive datatype. They present a syntax tree containing both expressions and declarations.

```
data Expr =
      Const Int
    | Add   Expr
    | Mul   Expr
    | EVar  String
    | Let   Expr
data Decl =
      String := Expr
    | Seq Decl Decl
```

  In this example the *Expr* datatype uses the *Decl* datatype and vice versa.

- *Nested datatypes* are parametrized recursive datatypes that have one or more recursive positions in which the type parameter changes. An example of a nested datatype is the *perfect tree*[Hin00], a tree that grows in size exponentially at every recursive position.

  ```
  data PerfectTree α =
      PerfLeaf α
    | PerfNode (PerfectTree (α, α))
  ```

  Because the *PerfectTree* uses a pair in the type index in the recursive position, every deeper level contains twice as many elements as their previous level.

- *Generalized algebraic datatypes*, or *GADTs*, have the possibility to add extra indices to the datatype that can be specialized on a per-constructor basis. These indices can be used to encode type invariants can limit the recursive nesting. GADTs with indices are called indexed datatypes or higher order datatypes. A common example of an indexed datatype is the *Vector* type, a sequence that encodes its length in the type. The *Vector* datatype uses Peano style natural numbers at the type level, which is either a zero or a successor of another natural number. For conciseness, natural number at the type level are typeset as true numbers.

  ```
  data 0
  data α + 1
  ```

  Just like Haskell lists, the *Vector* type has two constructors: *Nil* and *Cons*. The *Nil* can be used to construct an empty *Vector* with index 0, the *Cons* can be used to prepend one value to a *Vector* increasing its length with one.

  ```
  data Vector i α where
    Nil  ::                     Vector 0       α
    Cons :: α → Vector i α → Vector (i + 1) α
  ```

These three examples show that regular datatypes are not the only commonly used container data types. Unfortunately, the generic programming techniques introduced in the previous chapters do not allow mutually recursive datatypes, nested datatypes and indexed datatypes to be used with our storage framework. In this chapter we will extend the framework to allow using indexed datatypes, including indexed GADTs, as the persistent container datatypes. The global architecture of the system remains the same, but, as we will see, we have to rewrite almost all individual components to allow the use of indexed datatypes.

In this chapter we will only focus on explicitly indexed datatypes using GADTs. We will not discuss mutually recursive datatypes and nested datatypes. However, we will as an example show how to rewrite a nested finger tree[HP06] datatype to a single indexed GADT. The GADT will have the same invariants as the original nested finger tree type as presented by Hinze et al.

Because generalized algebraic datatypes are a generalization of the algebraic datatypes in Haskell, all nested datatypes can be written down as a GADT. By making the nested behaviour of nested datatypes explicit using a type variable in a GADT our framework should also be applicable to other nested datatypes.

Rodriguez et al. [RHLJ08] show how to encode a family of mutually recursive datatypes as an indexed GADT in order to perform generic programming over this family. This observation indicates that the persistence framework for indexed datatypes we are about to introduce will also work for families of mutually recursive datatypes. However, the families of datatypes have to be written down in a slightly different encoding.

6.1 Higher order fixed points

We start at the same place as we did for the framework for regular recursive datatypes: we introduce an annotated fixed point combinator. The higher order fixed point combinator is very similar to the regular fixed combinator although it is parametrized with an additional type index called *ix*. The *ix* is used as an additional type variable to the container structure f, either directly or indirectly through the annotation type α. The higher order annotated fixed point combinator will be called γ_α. In this chapter we will commonly prefix types and function names with an additional H or h to make clear we are dealing with the higher order variant of the type or function.

$$
\begin{aligned}
&\textbf{data}\ \gamma_\alpha\ \alpha\ f\ ix\\
&\quad = In^h_\alpha\ \{out^h_\alpha :: \alpha\ f\ (\gamma_\alpha\ \alpha\ f)\ ix\}\\
&\quad \mid In^h_f\ \{out^h_f :: \quad f\ (\gamma_\alpha\ \alpha\ f)\ ix\}
\end{aligned}
$$

Now recall the original annotated fixed point combinator:

$$
\begin{aligned}
&\textbf{data}\ \mu_\alpha\ \alpha\ f\\
&\quad = In_\alpha\ \{out_a :: \alpha\ f\ (\mu_\alpha\ \alpha\ f)\}\\
&\quad \mid In_f\ \{out_f :: \quad f\ (\mu_\alpha\ \alpha\ f)\}
\end{aligned}
$$

At first sight, this higher order version looks very similar to the regular fixed combinator, the only obvious addition is the index parameter *ix*. But because of to this index the kinds of the other type variables become more complicated. Assuming a kind $\star$ for *ix*, the type kind of variable f changes so that every $\star$ is replaced with an $\star \to \star$.

$$\begin{array}{ll} \star & \to \star \\ (\star \to \star) & \to \star \to \star \end{array}$$

Figure 6.1: Kind of *ix* for μ_α vs. kind of *ix* for γ_α.

Because the kind of f grows, the kind of the type variable α grows with it. Again, every $\star$ is replaced by a $\star \to \star$.

$$\begin{array}{llllll} (& \star & \to \star &) \to & \star & \to \star \\ ((& \star \to \star) & \to \star \to \star &) \to & (\star \to \star) & \to \star \to \star \end{array}$$

Figure 6.2: Kind of f for μ_α vs. kind of *ix* for γ_α.

This change in kinds has a lot of impact on the rest of our framework. Introducing one additional type index forces us to changes almost all the types in our system. We start by defining the indexed identity annotation.

$$\textbf{newtype}\ Id^h\ f\ \alpha\ ix = Id^h\ \{unId^h :: f\ \alpha\ ix\}$$

As the type shows, the identity annotation also takes the additional type index *ix* and uses this to parametrize the indexed functor f. Although this annotation is very similar to the regular identity annotation defined in section 2.2, it is very clear why we cannot reuse the original type.

Now we can create a annotation free fixed point combinator again by parametrizing the annotated fixed point with the identity annotation.

$$\textbf{type}\ \gamma\ f\ ix = \gamma_\alpha\ Id^h\ f\ ix$$

6.2 Finger tree as GADT

To illustrate the usage of the higher order fixed point combinator we will now model a finger tree data type as a indexed GADT. The finger tree is a purely functional data structure that can be used to model an abstract sequence with very interesting runtime behaviour. Hinze and Paterson[HP06] show how to implement a finger tree in Haskell using a nested datatype. The datatypes the authors describe looks more or less[1] like the following:

```
data Node α = Node₂ α α | Node₃ α α α
data Digit α = Digit₁ α | Digit₂ α α | Digit₃ α α α | Digit₄ α α α α
data FingerTree α
   = Empty
   | Single α
   | Deep (Digit α)
          (FingerTree (Node α))
          (Digit α)
```

The *FingerTree* datatype can be seen as a spine structure containing zero, one or two 2-3 trees per spine node. In this example 2-3 trees are trees that branch with a factor of 2 or a factor of 3 in their recursive positions. To allow a bit more flexibility at the root nodes of the 2-3 trees the *Digit* type is used to allow a branching factor between 1 and 4. Hinze and Paterson show that flexing the branching factor on the top-level of the 2-3 trees makes it easier to write algorithms and efficiency proofs for finger trees, but proof this does not negatively influence the asymptotic running time of the algorithms. Finger trees containing zero or one 2-3 trees must be created with the *Empty* or *Single* constructors and have only one spine node. Larger finger trees containing more spine nodes are constructed using the *Deep* constructor. The *Deep* constructor takes two 2-3 trees – with a *Digit* root node – and take one extra spine node for the rest of the tree. At every recursive position the value type changes by surrounding it with one *Node* level. The usage of the *Node* type for the recursion implicitly encodes the depth of the 2-3 trees at every level of the spine grows with one.

The paragraph above shows that the nested finger tree datatypes encodes some important invariants about the structure in its type. Before we will try to encode the same structure as an indexed GADT we will extract an explicit specification from the nested datatype definition above.

After identifying these structural properties we create a specification for the definition of finger trees. We will use this specification to encode finger trees as a GADT.

1. A finger tree contains *spine nodes*, *digit nodes*, *2-3 nodes* and *values*.

[1]In their paper Hinze and Paterson use a slightly less constraint type in which they encode the digit as a Haskell list datatype, with the implicit invariant this list has length greater than or equal to 1 and less than or equal to 4. We use the *Digit* datatype with four constructors to be a bit more explicit.

2. A spine node is either an *Empty* node, a *Single* node containing one 2-3 tree or a *Deep* node containing two 2-3 trees and a tail finger tree. The spine nodes form a linearly linked list.

3. The first spine node in a non-singleton finger tree only contains two *single-level* 2-3 trees. The second spine node only contains two *two-level* 2-3 trees, the third level stores two *three-level* 2-3 trees and so on. A quick calculation shows that every spine node in a finger tree, except for the last terminating node, contains a maximum of $2 * 4 * 3^n$ values, where n is the depth index of the node.

4. A 2-3 tree is either a single value of type α, a branch node with 2 or 3 sub trees or a root node which is called a digit.

5. Digits have a branching factor of 1, 2, 3 or 4. Digits are the root nodes of 2-3 tree and can only contain nodes or values.

We can now try to capture the specification, including the invariants between the different nodes, in an indexed GADT. Our finger tree GADT will contain a single product type as the index. The product contains two components, one for the kind of finger tree node and one for the encoding of the three depths. We first define three phantom types for indices to indicate a spine node, a digit node and a 2-3 tree node.

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

The depth encoding will be written down as a subscript type level natural number, similar to the index of the *Vector* datatype above. The depth index forms the second component of the index tuple.

```
type Sp_i  = (Sp, i)
type Dg_i = (Dg, i)
type Nd_i = (Nd, i)
```

Now we can define the finger tree GADT that encodes all the invariants from the specification.

```
data Tree (β :: ⋆) (f :: ⋆ → ⋆) :: ⋆ → ⋆ where
  Empty ::                                   Tree β f Sp_{i+1}
  Single :: f Dg_{i+1}                     → Tree β f Sp_{i+1}
  Deep   :: f Dg_{i+1} → f Sp_{i+2} → f Dg_{i+1}   → Tree β f Sp_{i+1}
  Digit_1 :: f Nd_i                         → Tree β f Dg_{i+1}
  Digit_2 :: f Nd_i → f Nd_i                → Tree β f Dg_{i+1}
```

$$
\begin{array}{lll}
\mathit{Digit}_3 & :: f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i & \to \mathit{Tree}\ \beta\ f\ \mathit{Dg}_{i+1} \\
\mathit{Digit}_4 & :: f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i & \to \mathit{Tree}\ \beta\ f\ \mathit{Dg}_{i+1} \\
\mathit{Node}_2 & :: f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i & \to \mathit{Tree}\ \beta\ f\ \mathit{Nd}_{i+1} \\
\mathit{Node}_3 & :: f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i \to f\ \mathit{Nd}_i & \to \mathit{Tree}\ \beta\ f\ \mathit{Nd}_{i+1} \\
\mathit{Value} & :: \beta & \to \mathit{Tree}\ \beta\ f\ \mathit{Nd}_0
\end{array}
$$

The type parameter β is used for the value type that is stored in the sequence. The finger tree has an explicit type parameter f for the recursive positions, just like the binary tree example for the regular recursive datatypes. Both the finger tree datatype itself and the type variable for the recursive positions have kind $\star \to \star$ and expect a type index. Every constructor produces a finger tree with its own specific index and can enforce a constraint between this index and the index of the recursive positions.

The definition shows that the *Empty*, *Single* and *Deep* constructors all create spine nodes with a non-zero depth index. A *Single* spine node contains one *Digit* root node of a 2-3 tree with exactly the same non-zero depth as the spine node encodes. This constraint is enforced in the index parameter for the recursive structure f. The *Deep* spine node contains two *Digit* root nodes for two 2-3 trees, also with the same depth index. The tail finger tree stored in the *Deep* constructor's second argument must also be a spine node with a depth index one larger than the current spine node. This enforces the tail to store more values than the head of a finger tree as specified in point 3 of the specification.

There are four different constructors for creating digit nodes, one for each arity. The digit constructors all take sub trees with the *Nd* index at the recursive positions, which can mean both 2-3 branching nodes or plain values. The digit nodes all have a depth index one larger than their sub trees.

Like digit nodes, the 2-3 node constructors Node_2 and Node_3 take sub trees with the *Nd* index at the recursive positions. This means a 2-3 branching node again or plain values. Also, the depth index is one larger than the index of their sub trees. A value is special kind of node that always has index zero. It contains a value of type β which is the value type parameter of the finger tree datatype.

This indexed GADT encodes all the properties from our specification into a single structure. We can now use the higher order fixed point combinator to tie the knot and create a recursive data structure again by filling in the f type parameter.

$$\textbf{type}\ \mathit{FingerTree}_\alpha\ \alpha\ \beta = \gamma_\alpha\ \alpha\ (\mathit{Tree}\ \beta)\ \mathit{Sp}_1$$

So, an annotated finger tree with annotation type α that stores values of type β. A finger tree root node always is a spine node with depth index one.

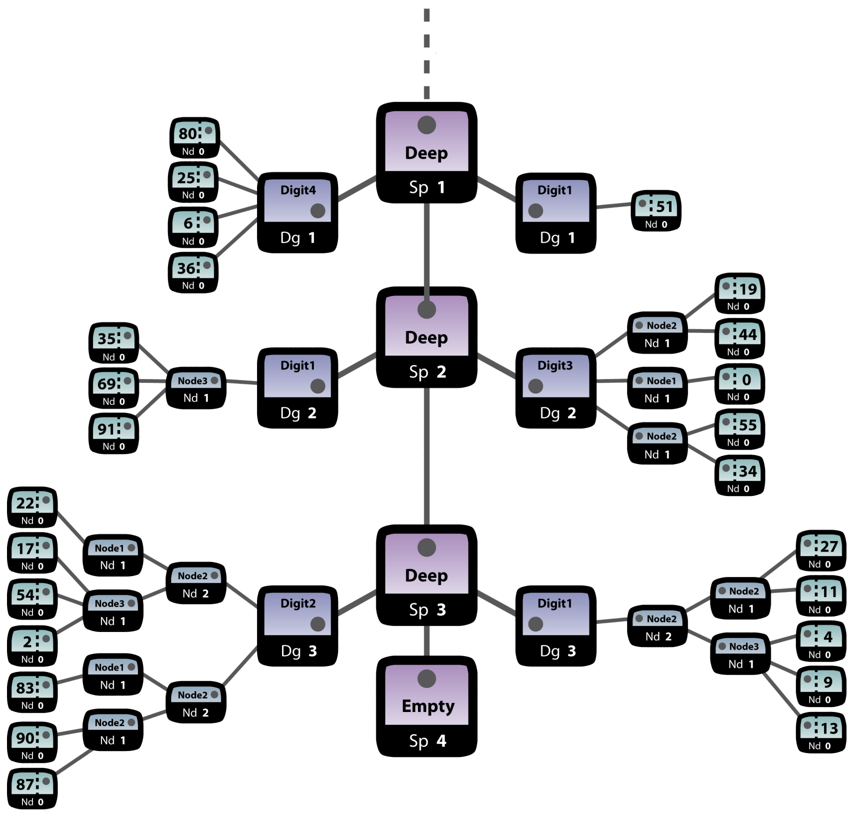

Figure 6.3: Example finger tree. This image shows the structure of finger trees and the positioning of spine nodes, digits, 2-3 nodes and value. The black boxes at the bottom of each node show the type level index.

Now we can also define a bunch of additional type synonyms to simplify working with our finger trees.

$$\begin{array}{l}
\textbf{type}\ \mathit{Node}\ \alpha\ \beta\ i = \gamma_\alpha\ \alpha\ (\mathit{Tree}\ \beta)\ \mathit{Nd}_i \\
\textbf{type}\ \mathit{Value}\ \alpha\ \beta\ \ \ = \gamma_\alpha\ \alpha\ (\mathit{Tree}\ \beta)\ 0 \\
\textbf{type}\ \mathit{Digit}\ \alpha\ \beta\ i = \gamma_\alpha\ \alpha\ (\mathit{Tree}\ \beta)\ \mathit{Dg}_{i+1} \\
\textbf{type}\ \mathit{Spine}\ \alpha\ \beta\ i = \gamma_\alpha\ \alpha\ (\mathit{Tree}\ \beta)\ \mathit{Sp}_{i+1}
\end{array}$$

6.3 Higher order traversals

In the section about fixed point combinators for regular datatypes we have derived *Functor* and *Traversable* instances for our binary tree example datatype. These traversal functions allow us to perform a restricted form of generic programming over our datatypes. Generic traversals form an essential part of our generic persistence framework. Unfortunately, these type classes cannot be used for our higher order datatypes like our finger tree GADT. In order to perform generic traversal over indexed datatypes we have to introduce higher order variant of these three type classes.

6.3.1 Functor type class

Ghani and Johann[GJ07] describe how to create a type class for higher order functors. The functorial map function works on indexed datatypes.

```
class Functor^h h where
  hfmap :: (∀ix.α ix → β ix) → ∀ix.h α ix → h β ix
```

We define a derived higher order functor type class *Functor*^{p} that gets an additional proof object ϕ. This object proves the index is an inhabitant of a specific *family*. This allows us to write functor instances that work not for all type indices, but for an explicitly limited set.

```
class Functor^p φ h where
  pfmap :: ( ∀ix.φ ix →   α ix →   β ix)
        → ∀ix.φ ix → h α ix → h β ix
```

When we compare both the *hfmap* and the *pfmap* signatures with the original *fmap* signature we clearly see the pattern:

```
fmap  :: (            α    → β   ) →             f α    → f β
hfmap :: (∀ix.        α ix → β ix) → ∀ix.        h α ix → h β ix
pfmap :: (∀ix.φ ix → α ix → β ix) → ∀ix.φ ix → h α ix → h β ix
```

In the next section we will show how to create an finger tree instances for the *Functor*^{p} type class and see how the proof object can be used to limit the set of possible indices.

6.3.2 Finger tree functor instance

The *Foldable*^{p} type class forces us to make explicit the index family we want to reason about, before we can create an instance for the finger tree datatype. We construct a GADT that serves as a proof object that proves that a certain index is actually a possible index for our *Tree* GADT. We will indicate proof types and proof values with a postfix ϕ.

```
data Tree_φ :: ⋆ → ⋆where
  Sp_φ   :: Nat_φ c → Tree_φ Sp_{c+1}
```

```
$Dg_\phi \;\; :: Nat_\phi\ c \rightarrow Tree_\phi\ Dg_{c+1}$
$Nd_\phi \;\; :: Nat_\phi\ c \rightarrow Tree_\phi\ Nd_{c+1}$
$Nd_{0\phi} :: \qquad\qquad\;\; Tree_\phi\ Nd_0$
```

This proof type can be used to show that for every natural number there is a *spine*, *digit* and *node* index for the successor of that natural number. Note that the structure of the $Tree_\phi$ proof indices is very similar to the structure of the finger tree indices. For this proof object we also need a natural number proof.

```
data $Nat_\phi :: \star \rightarrow \star$ where
  $\phi_0 \;\;\; :: \qquad\qquad\; Nat_\phi\ 0$
  $\phi_{i+1} :: Nat_\phi\ n \rightarrow Nat_\phi\ (n+1)$
```

This datatype can be used to construct a value for every natural number index. Again, the structure of the proof type follows the structure of the natural numbers.

No we can create a $Functor^p$ instance in which we can pattern match on both the proof object and the finger tree constructor. Because the type signature of *pfmap* parametrizes the proof $Tree_\phi$ and the constructor $Tree\ \beta$ with the same index, the compiler should be able to see that there are only a limited set of proof/constructor combinations possible.

The $Functor^p$ instance for finger trees becomes:

```
instance $Functor^p\ Tree_\phi\ (Tree\ \beta)$ where
  $pfmap\ f\ \phi\ h$ =
    case $(\phi, h)$ of
      $(Sp_\phi\ \_,\quad Empty)$            $\rightarrow Empty$
      $(Sp_\phi\ p,\quad Single\ \alpha)$         $\rightarrow Single\ (f\ (Dg_\phi\ p)\ \alpha)$
      $(Sp_\phi\ p,\quad Deep\ \alpha\ c\ \beta)$       $\rightarrow Deep\quad (f\ (Dg_\phi\ p)\ \alpha)$
                                          $(f\ (Sp_\phi\ \phi_{i+1})\ c)$
                                          $(f\ (Dg_\phi\ p)\ \beta)$
      $(Dg_\phi\ \phi_{i+1}, Digit_1\ \alpha)$         $\rightarrow Digit_1\ (f\ (Nd_\phi\ p)\ \alpha)$
      $(Dg_\phi\ \phi_{i+1}, Digit_2\ \alpha\ \beta)$       $\rightarrow Digit_2\ (f\ (Nd_\phi\ p)\ \alpha)$
                                          $(f\ (Nd_\phi\ p)\ \beta)$
      $(Dg_\phi\ \phi_{i+1}, Digit_3\ \alpha\ \beta\ c)$     $\rightarrow Digit_3\ (f\ (Nd_\phi\ p)\ \alpha)$
                                          $(f\ (Nd_\phi\ p)\ \beta)$
                                          $(f\ (Nd_\phi\ p)\ c)$
      $(Dg_\phi\ \phi_{i+1}, Digit_4\ \alpha\ \beta\ c\ d)$ $\rightarrow Digit_4\ (f\ (Nd_\phi\ p)\ \alpha)$
                                          $(f\ (Nd_\phi\ p)\ \beta)$
                                          $(f\ (Nd_\phi\ p)\ c)$
                                          $(f\ (Nd_\phi\ p)\ d)$
      $(Nd_\phi\ \phi_{i+1}, Node_2\ \alpha\ \beta)$       $\rightarrow Node_2\ (f\ (Nd_\phi\ p)\ \alpha)$
                                          $(f\ (Nd_\phi\ p)\ \beta)$
      $(Nd_\phi\ \phi_{i+1}, Node_3\ \alpha\ \beta\ c)$     $\rightarrow Node_3\ (f\ (Nd_\phi\ p)\ \alpha)$
                                          $(f\ (Nd_\phi\ p)\ \beta)$
                                          (f (Nd_φ p) c)
(Dg_φ φ_0,  Digit_1 α)        → Digit_1 (f Nd_0φ α)
(Dg_φ φ_0,  Digit_2 α β)      → Digit_2 (f Nd_0φ α)
                                          (f Nd_0φ β)
(Dg_φ φ_0,  Digit_3 α β c)    → Digit_3 (f Nd_0φ α)
                                          (f Nd_0φ β)
                                          (f Nd_0φ c)
(Dg_φ φ_0,  Digit_4 α β c d)  → Digit_4 (f Nd_0φ α)
                                          (f Nd_0φ β)
                                          (f Nd_0φ c)
                                          (f Nd_0φ d)
(Nd_0φ,     Value α)          → Value  α
```

To allow the *pfmap* function for this instance to apply the map function f to all the sub trees, we need to construct an appropriate proof again. We can do this by unpacking the input proof, repacking this with the right constructor and passsing it to the function f. This works out exactly right because the proof type follows the same index structure as our finger tree.

This *Functor*^{p} instance allows us to map a function over one level of recursive positions of the finger tree GADT. We can use the proof object to distinguish between different positions in the structure. The *Functor*^{p} instances will form the basis of generic traversals over higher order datatypes with restricted families of indices.

6.3.3 Traversable type class

Besides the higher order *functor* instance we can also make a higher order *traversable* instance, allowing us to perform effectful traversals. First we define a *Traversable*^{p} type class similar to the *Functor*^{p} type class, we also use a proof object ϕ to restrict the family of indices.

```
class Functor^p φ h ⇒ Traversable^p φ h where
  ptraverse :: Applicative f
            ⇒ (∀ix.φ ix →    α ix → f (  β ix))
            →  ∀ix.φ ix → h α ix → f (h β ix)
```

So if we are provided an effecful computation for the element type α *ix* in some *Applicative* – or possibly monadic – context f, the *ptraverse* function should be able to apply this to all elements of the structure $h\ \alpha$ *ix*.

6.3.4 Finger tree traversable instance

The *Traversable*^{p} instance for the finger tree GADT follows the same pattern as the regular *Traversable* instance. But this indexed variant requires us to pattern match on- and recursively supply the proof objects. This is similar to the *Functor*^{p} instance.

```
instance Traversable^p Tree_φ (Tree α) where
  ptraverse f φ h =
    case (φ, h) of
      (Sp_φ _,     Empty)             → pure Empty
      (Sp_φ p,     Single α)          → pure Single ⊛ (f (Dg_φ p) α)
      (Sp_φ p,     Deep   α c β)      → pure Deep   ⊛ (f (Dg_φ p) α)
                                                    ⊛ (f (Sp_φ φ_{i+1}) c)
                                                    ⊛ (f (Dg_φ p) β)
      (Dg_φ φ_{i+1}, Digit_1 α)       → pure Digit_1 ⊛ (f (Nd_φ p) α)
      (Dg_φ φ_{i+1}, Digit_2 α β)     → pure Digit_2 ⊛ (f (Nd_φ p) α)
                                                     ⊛ (f (Nd_φ p) β)
      (Dg_φ φ_{i+1}, Digit_3 α β c)   → pure Digit_3 ⊛ (f (Nd_φ p) α)
                                                     ⊛ (f (Nd_φ p) β)
                                                     ⊛ (f (Nd_φ p) c)
      (Dg_φ φ_{i+1}, Digit_4 α β c d) → pure Digit_4 ⊛ (f (Nd_φ p) α)
                                                     ⊛ (f (Nd_φ p) β)
                                                     ⊛ (f (Nd_φ p) c)
                                                     ⊛ (f (Nd_φ p) d)
      (Nd_φ φ_{i+1}, Node_2 α β)      → pure Node_2 ⊛ (f (Nd_φ p) α)
                                                    ⊛ (f (Nd_φ p) β)
      (Nd_φ φ_{i+1}, Node_3 α β c)    → pure Node_3 ⊛ (f (Nd_φ p) α)
                                                    ⊛ (f (Nd_φ p) β)
                                                    ⊛ (f (Nd_φ p) c)
      (Dg_φ φ_0,   Digit_1 α)         → pure Digit_1 ⊛ (f Nd_0φ α)
      (Dg_φ φ_0,   Digit_2 α β)       → pure Digit_2 ⊛ (f Nd_0φ α)
                                                     ⊛ (f Nd_0φ β)
      (Dg_φ φ_0,   Digit_3 α β c)     → pure Digit_3 ⊛ (f Nd_0φ α)
                                                     ⊛ (f Nd_0φ β)
                                                     ⊛ (f Nd_0φ c)
      (Dg_φ φ_0,   Digit_4 α β c d)   → pure Digit_4 ⊛ (f Nd_0φ α)
                                                     ⊛ (f Nd_0φ β)
                                                     ⊛ (f Nd_0φ c)
                                                     ⊛ (f Nd_0φ d)
      (Nd_0φ,      Value α)           → pure (Value α)
```

both the higher order *Functor*^{p} and *Traversable*^{p} instances for our finger tree GADT, we can now start writing generic recursive traversals.

6.4 Higher order annotation classes

In our generic annotation framework for regular datatypes we have created three type classes to associate custom functionality with wrapping and unwrapping annotations. We have to do the same for our higher order annotation framework, but we cannot use the existing type classes due a clear type mismatch. In this chapter we extend the Ann_{In} and Ann_{Out} type classes to work with indexed data types and show how make instances for the higher order identity annotation.

First, recall the type signatures of the original annotation functions.

$$\begin{aligned}
&\textbf{type}\ In\ \ \ \ \ \ \alpha\ f\ m = f\ (\mu_\alpha\ \alpha\ f) \to m\ \ (\mu_\alpha\ \alpha\ f\ )\\
&\textbf{type}\ Out\ \ \ \alpha\ f\ m = \ \ \ \ \mu_\alpha\ \alpha\ f \ \ \to m\ (f\ (\mu_\alpha\ \alpha\ f))\\
&\textbf{type}\ InOut\ \alpha\ f\ m = (f\ (\mu_\alpha\ \alpha\ f) \to \ \ f\ (\mu_\alpha\ \alpha\ f))\\
&\qquad\qquad\qquad\quad \to \ \mu_\alpha\ \alpha\ f\ \to m\ \ (\mu_\alpha\ \alpha\ f\ )
\end{aligned}$$

We now define the three type signatures for the higher order query, producer and modifier functions. The type signatures already make clear the difference between annotation working on indexed types and our previous annotations working on regular recursive data structures.

$$\begin{aligned}
&\textbf{type}\ In^h\ \ \ \ \ \ \alpha\ h\ \phi\ m = \ \forall ix.\phi\ ix \to h\ (\gamma_\alpha\ \alpha\ h)\ ix \to m\ \ \ (\gamma_\alpha\ \alpha\ h\ \ ix)\\
&\textbf{type}\ Out^h\ \ \ \alpha\ h\ \phi\ m = \ \forall ix.\phi\ ix \to \ \ \ \gamma_\alpha\ \alpha\ h\ \ ix \to m\ (h\ (\gamma_\alpha\ \alpha\ h)\ ix)\\
&\textbf{type}\ InOut^h\ \alpha\ h\ \phi\ m = (\forall ix.\phi\ ix \to h\ (\gamma_\alpha\ \alpha\ h)\ ix \to \ \ \ h\ (\gamma_\alpha\ \alpha\ h)\ ix)\\
&\qquad\qquad\qquad\qquad\quad \to \forall ix.\phi\ ix \to \ \ \ \gamma_\alpha\ \alpha\ h\ \ ix \to m\ \ \ (\gamma_\alpha\ \alpha\ h\ \ ix)
\end{aligned}$$

All three function types take an annotation α, a higher order recursive structure h with explicit recursive positions, an index family proof object ϕ and some monadic context m.

The In^h type describes a producer function that takes a single unannotated node with fully recursive sub structures and wraps it with some annotation, possibly inside some effectful context. The functions can restrict the family of indices with the proof term ϕ. The Out^h type describes a query function that performs the dual task of the producer. It takes a fully annotated tree and unwraps the root annotation, returning a single unannotated node with fully annotated sub structures. The modifier function $InOut^h$ simply combines the query and producer functions in one step.

Now we introduce the three type classes that implement respectively a producer, a query and a modifier function.

$$\begin{aligned}
&\textbf{class}\ (Applicative\ m, Monad\ m) \Rightarrow Ann^h_{Out}\ \alpha\ h\ \phi\ m\ \textbf{where}\\
&\quad ann^h_{Out} :: Out^h\ \alpha\ h\ \phi\ m
\end{aligned}$$

$$\begin{array}{l}\textbf{class}\ (\mathit{Applicative}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{Ann}^h_{\mathit{In}}\ \alpha\ h\ \phi\ m\ \textbf{where}\\ \quad \mathit{ann}^h_{\mathit{In}} :: \mathit{In}^h\ \alpha\ h\ \phi\ m\end{array}$$

Again, we create a modifier type class that has a default implementation in terms of the query and producer function.

$$\begin{array}{l}\textbf{class}\ (\mathit{Ann}^h_{\mathit{Out}}\ \alpha\ h\ \phi\ m, \mathit{Ann}^h_{\mathit{In}}\ \alpha\ h\ \phi\ m) \Rightarrow \mathit{Ann}^h_{\mathit{IO}}\ \alpha\ h\ \phi\ m\ \textbf{where}\\ \quad \mathit{ann}^h_{\mathit{IO}} :: \mathit{InOut}^h\ \alpha\ h\ \phi\ m\\ \quad \mathit{ann}^h_{\mathit{IO}}\ f\ \phi = \mathit{ann}^h_{\mathit{In}}\ \phi \circ f\ \phi \bullet \mathit{ann}^h_{\mathit{Out}}\ \phi\end{array}$$

The instances for the Id^h annotation are as simple as wrapping and unwrapping the constructor.

$$\begin{array}{l}\textbf{instance}\ (\mathit{Applicative}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{Ann}^h_{\mathit{Out}}\ \mathit{Id}^h\ h\ \phi\ m\ \textbf{where}\\ \quad \mathit{ann}^h_{\mathit{Out}}\ \_\ (\mathit{In}^h_{\alpha}\ (\mathit{Id}^h\ f)) = \mathit{return}\ f\\ \quad \mathit{ann}^h_{\mathit{Out}}\ \_\ (\mathit{In}^h_{f}\ f) \qquad\ = \mathit{return}\ f\\ \textbf{instance}\ (\mathit{Applicative}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{Ann}^h_{\mathit{In}}\ \mathit{Id}^h\ h\ \phi\ m\ \textbf{where}\\ \quad \mathit{ann}^h_{\mathit{In}}\ \_ = \mathit{return} \circ \mathit{In}^h_{\alpha} \circ \mathit{Id}^h\\ \textbf{instance}\ (\mathit{Applicative}\ m, \mathit{Monad}\ m) \Rightarrow \mathit{Ann}^h_{\mathit{IO}}\ \mathit{Id}^h\ h\ \phi\ m\end{array}$$

And from these type classes we can simply derive the two functions to fullly annotate or fully strip all annotations from the top of a tree.

$$\begin{array}{l}\mathit{fully}^h_{\mathit{In}}\ ::\ (\mathit{Ann}^h_{\mathit{In}}\ \alpha\ h\ \phi\ m, \mathit{Traversable}^p\ \phi\ h)\\ \qquad\quad \Rightarrow \phi\ ix \to \gamma_{\alpha}\ \alpha\ h\ ix \to m\ (\gamma_{\alpha}\ \alpha\ h\ ix)\\ \mathit{fully}^h_{\mathit{In}}\ \phi\ (\mathit{In}^h_{f}\ f) = \mathit{ptraverse}\ \mathit{fully}^h_{\mathit{In}}\ \phi\ f \rightarrowtail \mathit{ann}^h_{\mathit{In}}\ \phi\\ \mathit{fully}^h_{\mathit{In}}\ \_\ \alpha \qquad\quad = \mathit{return}\ \alpha\end{array}$$

$$\begin{array}{l}\mathit{fully}^h_{\mathit{Out}}\ ::\ (\mathit{Ann}^h_{\mathit{Out}}\ \alpha\ h\ \phi\ m, \mathit{Traversable}^p\ \phi\ h)\\ \qquad\quad \Rightarrow \phi\ ix \to \gamma_{\alpha}\ \alpha\ h\ ix \to m\ (\gamma_{\alpha}\ \alpha\ h\ ix)\\ \mathit{fully}^h_{\mathit{Out}}\ \phi\ (\mathit{In}^h_{\alpha}\ f) = \mathit{ann}^h_{\mathit{Out}}\ \phi\ (\mathit{In}^h_{\alpha}\ f) \rightarrowtail \mathit{fmap}\ \mathit{In}^h_{f} \circ \mathit{ptraverse}\ \mathit{fully}^h_{\mathit{Out}}\ \phi\\ \mathit{fully}^h_{\mathit{Out}}\ \_\ \alpha \qquad\quad = \mathit{return}\ \alpha\end{array}$$

Although the types have changed, the annotation framework is very similar to the one for regular recursive data types. We can now use these type classes to implement a paramorphism for indexed datatypes

6.5 Higher order paramorphism

In this section we introduce paramorphic traversals for higher order datatypes. In order to express the algebras we define the lifted sum and product types, similar to the Haskell *Either* α β and (α, β) (tuple) types, but with an additional type index.

$$\textbf{data}\ (f + g)\ ix = L\ (f\ ix) \mid R\ (g\ ix)$$
$$\textbf{data}\ (f \times g)\ ix = (\times)\ \{fst^h :: f\ ix, snd^h :: g\ ix\}$$

Using the product type we can construct a higher order paramorphic algebra. Like the algebra for regular datatypes, this algebra should be able to destruct one node with the recursive results in the recursive positions to a some result value. This indexed algebra can use a proof object as a restriction on the index family. Both the input structure f and the output structure g are indexed with the same index type ix. Because this is a paramorphic algebra, the f structure contains both the recursive results and the fully annotated sub structures.

$$\textbf{type}\ \Psi_{h\alpha}\ \alpha\ \phi\ f\ g = \forall ix.\phi\ ix \to f\ (\gamma_\alpha\ \alpha\ f \times g)\ ix \to g\ ix$$

Not that this algebra contains the type variable α and can be used for annotated traversals.

Now we define the higher order annotated paramorphism that uses an algebra to recursively destruct the structure $\gamma_\alpha\ \alpha\ f\ ix$ into a value of type $g\ ix$. Because this is an annotated paramorphism the traversal uses the ann^h_{Out} method from the Ann^h_{Out} type class to unwrap the annotation, possibly in an effecful context m.

```
$hpara^m_\alpha$
  $:: (Ann^h_{Out}\ \alpha\ f\ \phi\ m, Traversable^p\ \phi\ f)$
  $\Rightarrow \Psi_{h\alpha}\ \alpha\ \phi\ f\ g \to \phi\ ix \to \gamma_\alpha\ \alpha\ f\ ix \to m\ (g\ ix)$
$hpara^m_\alpha\ \psi\ \phi = return \circ \psi\ \phi\ \bullet$
    $ptraverse\ (\lambda p\ x \to (\!|\ (x\times)\ (hpara^m_\alpha\ \psi\ p\ x)\ |\!))\ \phi \bullet ann^h_{Out}\ \phi$
```

todo: not laziness, not applicative, future work

6.6 Sum, product, concat, and contains algebras

To illustrate the usage of the higher order paramorphism we will define four example algebras for our finger tree datatype. All four algebras will be defined in terms of one generic algebra that converts all value in a finger tree into some monoid value and appends these together using the $\oplus$ operator. The functionality of this algebra is very similar to the *foldMap* function of the *Foldable* type class.

The monoid type we use as the result type will be a plain type without any indices. Because the algebra forces the result type to have an index as well, we have to explicitly ignore the index when dealing with simple Haskell types. We do this by introducing the constant functor K, that accepts a value type and an index type, but only uses the value type and ignores the index type.

```
    newtype $K\ h\ \alpha = K\ \{unK :: h\}$
```

We can now create a generic fold algebra that returns some monoid m in the constant functor.

```
    $foldm_{alg} :: Monoid\ m \Rightarrow (\beta \to m) \to \Psi_{h\alpha}\ \alpha\ \phi\ (Tree\ \beta)\ (K\ m)$
    $foldm_{alg}\ f\ \_\ h =$
      case $h$ of
        $Empty$             $\to K\ \varnothing$
        $Single\ \alpha$            $\to K\ (g\ \alpha)$
        $Deep\ \ \alpha\ \beta\ c$          $\to K\ (g\ \alpha \oplus g\ \beta \oplus g\ c)$
        $Digit_1\ \alpha$           $\to K\ (g\ \alpha)$
        $Digit_2\ \alpha\ \beta$          $\to K\ (g\ \alpha \oplus g\ \beta)$
        $Digit_3\ \alpha\ \beta\ c$        $\to K\ (g\ \alpha \oplus g\ \beta \oplus g\ c)$
        $Digit_4\ \alpha\ \beta\ c\ d$ $\to K\ (g\ \alpha \oplus g\ \beta \oplus g\ c \oplus g\ d)$
        $Node_2\ \alpha\ \beta$          $\to K\ (g\ \alpha \oplus g\ \beta)$
        $Node_3\ \alpha\ \beta\ c$        $\to K\ (g\ \alpha \oplus g\ \beta \oplus g\ c)$
        $Value\ \alpha$            $\to K\ (f\ \alpha)$
      where $g = unK \circ snd^h$
```

The first parameter of this algebra is a function that converts the values stored in the finger tree into some type m for which there is a *Monoid* instance available. Because this algebra is a paramorphic algebra the function can use both the recursive results and the original sub structures. Note that the $foldm_{alg}$ only uses the recursive results, which actually makes it a *catamorphism*.

The paramorphism also allows us to add an index type to the result value, because we do not use this we ignore the index using the K constructor. Unpacking the recursive result from the tuple and from the K type is done with the helper function g.

The *Monoid* type class allows us to be generic in the type we want to fold. By specializing the value type to *Int* and the output type to *Sum Int* or *Product Int*[2], we can create two algebras that respectively compute the sum and the product of a sequence containing integers.

```
    $sum_{alg} :: \Psi_{h\alpha}\ \alpha\ \phi\ (Tree\ Int)\ (K\ (Sum\ Int))$
    $sum_{alg} = foldm_{alg}\ Sum$
```

[2]The *Sum* and *Product* newtypes can be found in Haskell's *Data.Moinoid* package. They are used to specialize the way the ⊕ operators combines two numeric values, either with addition or with multiplication.

$product_{alg} :: \Psi_{h\alpha}\ \alpha\ \phi\ (Tree\ Int)\ (K\ (Product\ Int))$
$product_{alg} = foldm_{alg}\ Product$

Creating a fold algebra that concatenates all strings can simply be done by exploiting the default monoid instances for lists and parametrize $foldm_{alg}$ with the identity function.

$concat_{alg} :: \Psi_{h\alpha}\ \alpha\ \phi\ (Tree\ String)\ (K\ (String))$
$concat_{alg} = foldm_{alg}\ id$

The last example is a containment check. This algebra uses an equality check in combination with the *Any* monoid wrapper to check whether a certain value exists in the finger tree sequence.

$contains_{alg} :: Eq\ \beta \Rightarrow \beta \rightarrow \Psi_{h\alpha}\ \alpha\ \phi\ (Tree\ \beta)\ (K\ Any)$
$contains_{alg}\ v = foldm_{alg}\ (Any \circ (== v))$

These four algebras can now be lifted to true annotated traversals using the $hpara^{m}_{\alpha}$ function from the previous section. Note that in all four cases we supply an index proof to the paramorphism. This proof object contains the same index as the root of our finger tree, which is $Sp_{\phi}\ \phi_0$. After computing the result we unpack it from the constant functor K and from the monoid wrapper when needed. The derived functions are shown below.

$$\begin{aligned}
&sum :: Ann^{h}_{Out}\ \alpha\ (Tree\ Int)\ Tree_{\phi}\ m \\
&\quad \Rightarrow FingerTree_{\alpha}\ \alpha\ Int \rightarrow m\ Int \\
&sum\ h = (\!|\ (getSum \circ unK)\ (hpara^{m}_{\alpha}\ sum_{alg}\ (Sp_{\phi}\ \phi_0)\ h)\ |\!)
\end{aligned}$$

$$\begin{aligned}
&product :: Ann^{h}_{Out}\ \alpha\ (Tree\ Int)\ Tree_{\phi}\ m \\
&\quad \Rightarrow FingerTree_{\alpha}\ \alpha\ Int \rightarrow m\ Int \\
&product\ h = (\!|\ (getProduct \circ unK)\ (hpara^{m}_{\alpha}\ product_{alg}\ (Sp_{\phi}\ \phi_0)\ h)\ |\!)
\end{aligned}$$

$$\begin{aligned}
&concat :: Ann^{h}_{Out}\ \alpha\ (Tree\ String)\ Tree_{\phi}\ m \\
&\quad \Rightarrow FingerTree_{\alpha}\ \alpha\ String \rightarrow m\ String \\
&concat\ h = (\!|\ unK\ (hpara^{m}_{\alpha}\ concat_{alg}\ (Sp_{\phi}\ \phi_0)\ h)\ |\!)
\end{aligned}$$

$$\begin{aligned}
&contains :: (Eq\ \beta, Ann^{h}_{Out}\ \alpha\ (Tree\ \beta)\ Tree_{\phi}\ m) \\
&\quad \Rightarrow \beta \rightarrow FingerTree_{\alpha}\ \alpha\ \beta \rightarrow m\ Bool \\
&contains\ v\ h = (\!|\ (getAny \circ unK)\ (hpara^{m}_{\alpha}\ (contains_{alg}\ v)\ (Sp_{\phi}\ \phi_0)\ h)\ |\!)
\end{aligned}$$

These four algebras show that it does not take that much to implement simple catamorphisms, that compute values of simple Haskell types, over indexed data structures. In the next section we show a more complex example, the *cons* function that appends one item to the beginning of the finger tree sequence.

6.7 Prepend algebra

Two of the basic operations on finger trees as described in the paper by Hinze and Paterson are the *cons* and the *snoc* functions. These functions prepend or append one item to the sequence. Two seemingly simple functions. However, both functions are a bit involved because they have to preserve the nested structure of the finger tree. In this section we briefly show what is needed to write a paramorphic algebra for the *cons* function on our finger tree GADT.

It takes some work to work out the type correct type signature for the algebra of the *cons* function. We need to construct a paramorphism that takes as input the original annotated finger tree and computes a function that takes the node to prepend and results in the new finger tree. We do need a lot of type level helpers to ensure the paramorphism resulting in an prepend function conforms all the type level invariants encoded in the GADT.

First we define two type families to compute the first and the second type level component from our indexed product type. Both type level families simply project the left or right component and apply the original index to the result.

```
type family Fst α :: ⋆
type instance Fst ((α × β) ix) = α ix


type family Snd α :: ⋆
type instance Snd ((α × β) ix) = β ix
```

The prepend function that is computed by the paramorphic *cons* algebra needs itself to reason about the index types. We need a higher order function type that distributes an index to both the co- and contra variant positions.

```
infixr 1↠
data (↠) α β ix = F {unF :: α ix → β ix}
```

We define the # operator to apply an indexed function to an indexed argument.

```
(#) :: (α↠β) ix → α ix → β ix
(#) (F x) y = x y
```

The inductive structure of the GADT forces us to reason about the numeric index relation of two connected nodes. Computing the successor of a type level natural number can be done using the successor (written down as $n+1$)

datatype. We now introduce a predecessor type family (written down as $n - 1$) that decrements any non-zero type level natural with one.

$$\textbf{type family}\ \cdot -1\ \alpha$$
$$\textbf{type instance}\ (c + 1) - 1 = c$$

Now we introduce two type level functions, encoded as datatypes, that allow us to change the GADT index. The *N* datatype takes a container datatype and an index and returns the same container type but with a different index. The *N* type level function forces the result to have a node type by using the *Nd* phantom type as the first component of index tuple. A *Maybe* type is to indicate a possible empty result. The *N* data function keeps the depth index intact.

$$\textbf{newtype}\ N\ f\ ix = N\ (Maybe\ (f\ (Nd, Snd\ ix)))$$

The *D* type level function encodes a similar pattern, but decrements the depth index by one using the predecessor type family defined above.

$$\textbf{newtype}\ D\ f\ ix = D\ (f\ (Fst\ ix, (Snd\ ix) - 1))$$

We now have all the components to write down the type signature for the paramorphic *cons* algebra. The type is rather complicated because it encodes all invariants involved when appending an item to the head of the sequence.

$$\begin{aligned} cons_{alg} &:: tree \sim \gamma_{\alpha}\ \alpha\ (Tree\ \beta) \\ &\quad \Rightarrow \Psi_{h\alpha}\ \alpha\ Tree_{\phi}\ (Tree\ \beta) \\ &\qquad (N\ (D\ tree) \twoheadrightarrow tree \times N\ tree) \end{aligned}$$

Let try to explain what this type means.

To simplify the type signature a type variable *tree* is defined as a shortcut for a fully annotated finger tree structure with value of type β. A type equality is used to make this local definition. We have a paramorphic algebra that takes an annotated finger tree as input and computes a function from a finger tree *node* to a product of some finger tree and some finger tree *node*. The result is a product type because it returns both the new finger tree and possibly a overflow node that needs to be inserted one level deeper. So, if we push a node to the head of sequence it gets inserted in the left 2-3 tree of the first spine node. When this 2-3 tree is already full one node (sub-tree) is selected to be apppended to a spine node one deeper. The input node always has a depth index one less than depth index of the sub-tree being traversed by the algebra, indicated by the *D* type. The result tree index is unaffected, this makes sense: appending a node to a finger tree should return a finger tree with the same index.

Both the input and overflow part of the output of the computed functions are nodes, because only values can be inserted inserted into a finger tree and only 2-3 trees can overflow to a deeper level. The *N* type also encodes the optionality using the *Maybe* type, this allows the algebra to stop the insertion when no node overflows. Hinze and Paterson[HP06] prove that overflowing happens rarely and the amortized time for appending a value is still $O(1)$.

One of the constraints of the $cons_{alg}$ is that we can only insert nodes with a depth index one less than the tree it gets appended to. At the top level a finger tree is always a spine node with depth index one, as can be seen in the $FingerTree_\alpha$ type synonym. This means we can only insert nodes with depth index zero into a top level finger tree. This is exactly as we would expect: only *Value* nodes have index zero. The *D* type family can only be applied to non-zero indices. To proof to the compiler our input finger tree always has a non-zero index we parametrize the algebra with our $Tree_\phi$ proof object.

6.7.1 Prepend function

Now that we have shown the type signature of the $cons_{alg}$ function, we can lift it using the $hpara_\alpha^m$ function to a true prepend function. Hinze and Paterson name this left-biased prepend function with the left pointing triangle: $\triangleleft$

The prepend function takes a value x and appends this to the existing sequence xs. The output of running the algebra against the xs is not a simple value but a *function*. This function takes the original finger tree to the result finger tree. We apply this function to the value x to get back the result finger tree. But, because we are working with annotations, we have to manually annotate the new parts of this result sequence with the $fully_{In}^h$ function.

$$
\begin{aligned}
&(\triangleleft) :: Ann_{IO}^h\ \alpha\ (Tree\ \beta)\ Tree_\phi\ m \\
&\quad \Rightarrow \beta \rightarrow FingerTree_\alpha\ \alpha\ \beta \rightarrow m\ (FingerTree_\alpha\ \alpha\ \beta) \\
&x \triangleleft xs = \mathbf{do}\ fun \leftarrow hpara_\alpha^m\ cons_{alg}\ (Sp_\phi\ \phi_0)\ xs \\
&\qquad\qquad\qquad \mathbf{let}\ res = fst^h\ (fun\ \#\ (N \circ Just \circ D)\ (In_f^h\ (Value\ x))) \\
&\qquad\qquad\qquad fully_{In}^h\ (Sp_\phi\ \phi_0)\ res
\end{aligned}
$$

As can be seen in the definition of the function, we have to include a lot of boilerplate code to pack and unpack all the intermediate wrappers. These wrappers, mostly type level functions, were introduced to allow working with indexed datatypes.

This example shows it is still possible to abstract away from recursion when writing more complicated operations over indexed datatypes. Higher order algebras can be written that preserve all invariants encoded in a GATD. However, we cannot help to conclude that *it is very hard to do so*. Lots of helper type level wrappers and functions have to be defined, only to be able to express the types. These wrappers force our implementation to include more boilerplate code which makes it harder to write.

6.8 Persistent Annotation

In order to serialize indexed datatypes to disk we need to make the *Pointer* type 4.1 an instance of our higher order annotation type classes. All the types and type classes involved need to be lifted to the indexed level, meaning we can not reuse any code of our previous framework.

For example, we cannot use the regular *Binary* type class, because this class only works for non-indexed types. So we create a higher order $Binary^h$ class useful for serializing and deserializing index datatypes.

$$
\begin{array}{l}
\textbf{class}\ Binary^h\ \phi\ h\ \textbf{where} \\
\quad hput :: \phi\ ix \to h\ ix \to Put \\
\quad hget :: \phi\ ix \to Get\ (h\ ix)
\end{array}
$$

Now we need to create an $Binary^h$ instance for our finger tree. We only show the class header and skip the implementation, which is rather straightforward. Note that for the element type β we still require a regular *Binary* instance.

$$\textbf{instance}\ (Binary\ \beta, Binary^h\ Tree_\phi\ f) \Rightarrow Binary^h\ Tree_\phi\ (Tree\ \beta\ f)$$

The second problem we encounter is that we cannot reuse the *read* and *write* functions working on our storage heap. Also these functions have to be lifted to an *hread* and an *hwrite* accordingly.

$$
\begin{array}{ll}
hread & :: Binary^h\ \phi\ h \Rightarrow \phi\ ix \to Pointer\ (h\ ix) \to Heap_R\ (h\ ix) \\
hwrite & :: Binary^h\ \phi\ h \Rightarrow \phi\ ix \to h\ ix \to Heap_W\ (Pointer\ (h\ ix))
\end{array}
$$

Now we lift the pointer annotation to a higher order level by extending the kinds. Because Haskell has no *kind polymorphism* and we need the additional type parameter *ix* we cannot reuse the existing $Pointer_\alpha$ type.

$$
\begin{array}{lll}
\textbf{newtype} & Pointer^h_\alpha & (f :: (\star \to \star) \to \star \to \star) \\
 & & (\beta :: \star \to \star) \\
 & & (ix :: \star) \\
\quad = & Pointer^h_\alpha & \{unP :: Pointer\ (f\ \beta\ ix)\}
\end{array}
$$

And now we can give the Ann^h_{Out}, Ann^h_{In} and Ann^h_{IO} instances for the $Pointer^h_\alpha$ annotation in both the read and read-write heap contexts. The implementations are rather straightforward and are shown below.

$$\begin{aligned}
&\textbf{instance}\ Binary^h\ \phi\ (h\ (\gamma_\alpha\ Pointer^h_\alpha\ h)) \Rightarrow Ann^h_{Out}\ Pointer^h_\alpha\ h\ \phi\ Heap_R\ \textbf{where}\\
&\quad ann^h_{Out}\ \phi\ (In^h_\alpha\ (Pointer^h_\alpha\ h)) = hread\ \phi\ h\\
&\quad ann^h_{Out}\ \_\ (In^h_f\ \qquad\qquad\quad h\ ) = return\ h
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ Binary^h\ \phi\ (h\ (\gamma_\alpha\ Pointer^h_\alpha\ h)) \Rightarrow Ann^h_{Out}\ Pointer^h_\alpha\ h\ \phi\ Heap_W\ \textbf{where}\\
&\quad ann^h_{Out}\ \phi\ (In^h_\alpha\ (Pointer^h_\alpha\ h)) = liftR\ (hread\ \phi\ h)\\
&\quad ann^h_{Out}\ \_\ (In^h_f\ h\ \qquad\qquad\quad) = return\ h
\end{aligned}$$

$$\begin{aligned}
&\textbf{instance}\ Binary^h\ \phi\ (h\ (\gamma_\alpha\ Pointer^h_\alpha\ h)) \Rightarrow Ann^h_{In}\ Pointer^h_\alpha\ h\ \phi\ Heap_W\ \textbf{where}\\
&\quad ann^h_{In}\ \phi = fmap\ (In^h_\alpha \circ Pointer^h_\alpha) \circ hwrite\ \phi
\end{aligned}$$

$$\textbf{instance}\ Binary^h\ \phi\ (h\ (\gamma_\alpha\ Pointer^h_\alpha\ h)) \Rightarrow Ann^h_{IO}\ Pointer^h_\alpha\ h\ \phi\ Heap_W\ \textbf{where}$$

These instances require that the higher order fixed point combinator must also be an instance of the $Binary^h$ class.

$$\begin{aligned}
&\textbf{instance}\ Binary^h\ \phi\ (\alpha\ f\ (\gamma_\alpha\ \alpha\ f)) \Rightarrow Binary^h\ \phi\ (\gamma_\alpha\ \alpha\ f)\ \textbf{where}\\
&\quad hput\ \phi\ (In^h_\alpha\ f) = hput\ \phi\ f\\
&\quad hget\ \phi \qquad\quad\ = (\!|\ In^h_\alpha\ (hget\ \phi)\ |\!)
\end{aligned}$$

Again, this section makes clear what the implication are of moving away from regular datatypes to indexed datatypes. From the high level annotation instances to the lower level read and write actions to the type class for binary serialization, everything has to be lifted to the indexed level.

6.9 Putting it all together

In the previous section we have in parallel, both extended the persistence framework to index datatypes and developed an indexed finger tree type as an example. Now we have all the ingredients to show that this higher order framework actually works. In this concluding section we conclude the chapter by showing how to build up a finger tree of integers on disk and compute the sum of all values.

First we introduce a type synonym for our finger tree of integers. We define an *IntStore* to be a persistent finger tree of integers by using the $Pointer^h_\alpha$ annotation at the recursive positions.

```
type IntStore = $\gamma_\alpha$ Pointer$^h_\alpha$ (Tree Int) Sp$_1$
```

The *pempty* function will be used to produce an empty persistent finger tree.

```
pempty :: Heap$_W$ IntStore
pempty = ann$^h_{In}$ (Sp$_\phi$ $\phi_0$) Empty
```

By only specializing the type we lift the prepend function ($\triangleleft$) from section 6.7.1 to work on persistent finger trees. We see that the type signatures of our generic functions become simpler the more we specialize them.

```
pcons :: Int → IntStore → Heap$_W$ IntStore
pcons = ($\triangleleft$)
```

We now have all the ingredient to write a function that creates and stores a finger tree on disk. We use the *run* function from our heap context to open a database file and run a heap computation.

```
createStore :: FilePath → IO ()
createStore file = run file $
  do p ← pcons 6 ⤆ pcons 3 ⤆ pcons 4 ⤆ pempty
     storeRoot p
```

Both the *pcons* and the *pempty* function result in an persistent finger tree, which means a *Pointer* to a finger tree. Because the *pcons* function also takes a persistent finger tree as input we can compose these actions together. Because the actions are monadic we composes these with the right-to-left monadic bind operator (⤆). After producing the tree we store the pointer in the root node of the heap.

We will create a second function that reads a finger tree from disk and computes the total sum over all values. First we lift the *sum* function to work on persistent finger trees, also by only changing the type.

```
psum :: IntStore → Heap$_R$ Int
psum = sum
```

Now we can write a function that reads our previously stored finger tree from the root node and computes the sum using our lifted *psum* function.

```
computeSum :: FilePath → IO ()
computeSum file = run file $
  do o ← getRoot
     s ← psum o
     liftIO (print s)
```

These two functions both work on the persistent store and can now be run in two different sessions. A consecutive run of first the *createStore* and then the *computeSum* results in the expected total 13:

```
ghci> createStore "test.db"
ghci> computeSum "test.db"
13
```

This example is the proof we have been able to extend our persistent framework to work with indexed datatypes. By lifting all the types to an indexed level we were able to build up persistent higher order structures. When using our annotation framework we can use the indices to reason about the position in a higher order structure. The results of this chapter allow us to build type safe container datatypes that encode structural invariants in their types. Although the final result is very pleasing, we must admit that writing operations on indexed datatypes in Haskell can be a tedious job.

Chapter 7

Related work

7.1 Clean

In their paper *Efficient and Type-Safe Generic Data Storage* Smetsers, Van Weelden and Plasmeijer[SWP08] describe a library for generic storage for the programming language Clean. Like the framework proposed in the document, they also aim at generically mapping pure and functional algebraic data structures to a persistent data storage on disk. Using something similar as our storage heap – they call this *Chunks* – they are able to persist individual parts of the data structures on disk without reading or writing the entire collection.

The big difference in their approach 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 they entire data structure itself is stored in one single chunk. Destructive updates of individual record values can now be done efficiently without touching the entire collection. But for every change in the structure of the collection the chunk containing the data structure itself — they call this the *Root chunk* — has to be read in and written back as a whole.

Because of the different method of slicing the data they do not pose the same problems in lifting the pure structures to a persistent variant. Most container data type definitions already are polymorphic in the element values they store. This significantly simplifies their implementation while making the system less flexible.

To improve type safety of their framework these Clean developers include a type hash inside their chunks. When reading data back in this type will be checked for validity. This is a very interesting technique that might also be used for improving the type safety of our Haskell framework.

7.2 Relational Database Management Systems

There are several connectors available for Haskell to existing relational database management systems. These connectors allow developers to directly query relational databases using the SQL query language. This gives developers the complete power of databases, like fast querying, indexing, transactions etc. Examples of packages for connecting to database management systems are the general Haskell Database Connection[Goe09] and the binding to the SQLite database system[Ste09].

Other libraries build on top of these RDBMS bindings and allow marshalling of Haskell data generically. Using generic programming libraries these connectors allow mapping values of arbitrary Haskell types to database rows and vice versa. These system really add some power, because developers do not have to write the marshalling code manually.

7.3 Key/Value stores

todo: TODO

7.4 Binary serialization

Data.Binary[LK09], developed by Don Stewart, is a Haskell library that enables generic serialization of values of arbitrary data types to binary streams. The library uses type classes to be polymorphic in the values to be serialized/deserialized. Out of the box the library forces users to write the type class instances for their data types by hand, but luckily there are several ways this can be done generically.

The Haskell *derive*[O'R09] package can be used to derive Data.Binary (and more) instances automatically for custom data types. The package uses (GHC only) Template Haskell for the derivation process but produces portable Haskell 98 code that can be linked manually into your project.

Using generic programming libraries like *EMGM*[OHL06], *SYB*[LP03] and *MultiRec*[RHLJ08] it should possible to generically derive instances as well. The advantage is that this can be used as a library and does not require external inclusion into your project. It might be interesting to note that using the above mentioned generic programming libraries – of which there are quite a few [RJJ$^+$08] – it should be fairly easy to skip the Data.Binary library entirely and write the functions from scratch, as described in [JJ99] and [SWP08].

Binary serialization will be an essential part of our persistence framework should ideally be done generically for all possible data types. The libraries explained above can be used to take care of this.

7.5 Algebras

In their paper *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* Meijer, Fokkinga and Paterson[MFP91] show how certain morphisms from category theory can be used to manipulate values of recursive data types without explicitly going into recursion. By using algebras and coalgebras to describe what actions to perform at certain points in the algorithm, they only need a few basic function to handle the actual recursion. Morphisms like catamorphisms and paramorphisms can be used to destruct a data type into a new value and can be seen as consumer functions. Anamorphisms can be used to create values of a data type and can be used to create producer functions. By combining or restricting these morphisms most if not all[**?**] algorithms working on functional data structures can be written without explicit recursion.

This is a very well explored and common trick in functional programming that will also be extensively used in this project. By writing algebras for data type specific folds, the container data remain open for annotation.

7.6 Attribute Grammars

Attribute grammars can be seen as a way of writing algebras for catamorphisms. The attribute grammar system for Haskell[SPS98] allows users to write operations over data structures in an aspect oriented way without explicitly coding out the recursion.

The different aspects written down using this attribute grammar system can potentially be used as a recipe for query and modifier functions over our persistent data structures.

7.7 Iteratee based IO

In Haskell, being a lazy language, the use of lazy IO has been quite common. Reading and processing the entire contents of a file lazily means there is no need to read the entire file into memory before processing it. Lazy IO has the awkward property that apparently pure Haskell expression have real world side effect and are therefor considered to be impure and 'not in the spirit of Haskell'.

To solve this problem Oleg Kiselyov has been working on left-fold enumerator, also known as iteratee-IO[Kis09a]. Iteratee based IO uses *fold* like functions for high-performance, incremental IO without the property that the evaluation of pure code cause side effects. A library for iteratee based IO is now available for Haskell[Kis09b].

The ideas from his work can be used to avoid the same pitfall in this project. We should make sure that processing values originating from our storage heap in pure functions does not cause any effects.

7.8 Sharing / Garbage Collection

Lots of research and work [SPJ93; MHJPJ08] has been done in the field of garbage collection for pure and lazy functional programming languages like Haskell. Lots of these techniques should be applicable to storage heaps outside the conventional computer memory, but located on disk.

7.9 Software Transactional Memory

Software Transactional Memory, STM, is a technique that is used to allow transactional access to shared memory. In their paper *Composable Memory Transactions* Harris, Marlow, Peyton Jones and Herlihy[HHMPJ05] describe an implementation of transactional memory for Haskell on top of *Concurrent Haskell*[JGF96]. This STM implementation for Haskell is composable, which means that separate block of atomic code can be composed into one larger block of atomic code. These atomic blocks of code can be used to give separate threads/processes exclusive access to certain memory variables without the intervening of other threads.

STM can not only be used to manage atomic actions to shared resources in memory, the abstraction can also be lifted to manage other program resources. This concept is implemented by the *TCache*[Cor09] package implements this concept by using STM to provide a transactional cache to arbitrary persistent back-ends.

These ideas, or maybe existing implementations, can be used to allow transactional access to our persistent data structures when working with concurrent programs.

Chapter 8

Future Work

In this document we have seen how to use generic programming for recursive data type to build a persistent storage framework in Haskell. There are plenty of opportunities for extension of the framework. This chapter gives a quick enumeration of topics that can be used to extend or improve this project.

8.1 Encoding type hashes into serialized data

This framework takes type values from the pure and type safe Haskell world and saves an binary representation to a flat file, this way loosing all type information. This is safe when only one single invocation of one single application touches the data, because the Haskell type system can be used to enforce this.

Guaranteeing type safety in between session is much harder to achieve. This problem can be solved by storing a simple cryptographic hash of the type of the data together with the data itself. When reading the data back in this hash can be used to verify whether the data is what is to be expected. A similar technique for the programming language Clean has been described in [SWP08].

8.2 Nested data structures

It is hard to predict whether it is very easy to extend the framework to allow the persistence of nested data types. It requires some research to figure out what adaptations, if at all, are needed to be able to store, for example, a binary tree of lists of records. In order to store nested data type we are at least in the need of a generic programming library that can handle this. See [GJ07].

8.3 Incremental folds

todo cite: sean on his work

Catamorphisms, or the more general paramorphisms, can be used to describe operations over recursive data structures. Techniques exist to cache intermediate results of these operations as an annotation at the recursive positions of the data structure. This prevents these algorithms form having to recompute a new value for the entire data structure when only certain parts of the structure change.

These caching operations are called incremental folds and could easily projected to the persistent data structures and saved together with non-recursive nodes. This would allow users to have very efficient queries over long-lived and incrementally changing data structures.

8.4 Using attribute grammars to write queries

The attribute grammar system for Haskell[SPS98] can be used as a DSL to describe algebras for catamorphisms. The current system produces an entire application that creates the data types, produces the catamorphisms and run the algberas using these catamorphisms. It would be very useful if could merely abstract the algebras from the system and use these to write queries over the persistent data structures.

Recent work has shown that it also possible to create a first-class attribute grammar system for Haskell. It might possibly be easier to use this system to write the algebras.

8.5 Parallel access to independent sub structures

The transactional cache described above will be used to guarantee that no two concurrent threads can alter the same tree at the same moment. Some recursive data structures though can have independent recursive sub structures to which concurrent access can be safe. It might be interesting to research if we can use the information about the recursive structure of our data types to allow safe concurrent access to independent sub structures.

8.6 Aspect oriented storage framework

Make the storage heap extendible in a nice way to allow custom variations on how data is saved. This should allow users to encrypt or compress individual data blocks. This extensions should also be able to incorporate the type hashes as described above.

8.7 Fixed point based optimizations

One single IO operation can generally be performed more efficiently than a large collection of smaller IO operations. When executing operations over large parts of a persistent structure it might be desirable to load a block of nodes into memory at once and perform a partial operation on this in-memory.

By using fixed point information as a measurement of locality in our data structures we might be able to store records that are near to each other in the same IO block. Operation which rely on the compositional structure of the data, like algebraically defined catamorphisms, might significantly benefit from such an optimization.

Chapter 9

Conclusion

In this document we have described a storage framework for algebraic data types in Haskell. Using the advanced Haskell type system in combination with generic programming techniques we have build a generic annotation system for recursive data structures. Annotation can be used to associate custom functionality with the construction, destruction and modification of recursive data structures. We have seen that using an identity annotation yields a data structures that runs in-memory. Switching the identity annotation with a pointer annotation yields a data structure that run on a storage heap on disk.

Haskell's powerful type system has allowed us to create a very clean separation between the storage layer and the persistent data structures. By writing operations on recursive data structure as algebras that abstract away from recursive we are able to plug-in custom actions. This allows us to write data structures that can transparently be stored on a persistent disk without writing explicit I/O code. The absence of I/O specific code makes it much more easy to reason about the correctness of the framework.

Although writing operations on recursive datatypes as algebras and coalgebras is a common functional programming technique that has been well described in literature, writing algebraic computations in practice can be hard. It requires a slight paradigm shift to abstract away from recursion, but we think it is worth the benefit.

The final interface to the user is very natural and comes close to what one might expect from working with in-memory data structures. The only drawback is that all operations are lifted to work in a monadic context. This context forces us to write composition of operations using monadic combinators instead of regular function composition. This limitation is very common when generically lifting functionality to different computational contexts in Haskell.

The annotation framework described in the first five chapters makes the assumption that the recursive datatypes are regular recursive datatypes. Datatypes making use of higher order recursion cannot be used. Extending the system to also work for indexed datatypes has showed not to be particularly hard, but did require almost duplicating the entire framework. Because the mayor type mismatch all the involved helper datatypes and type classes have to reimplemented with slightly different types. Unfortunately we do not know of a way to build an abstraction layer hiding the differences between regular datatypes and higher order datatypes. It also not clear whether this project could benefit from such an abstraction.

Although generic programming for higher order datatypes has been well covered in literature is not always easy. Especially the combination of higher order algebras for annotated recursive operations is far more involved than working with the regular recursive datatypes. The implementations of higher order operations might require additional boilerplate code for proof terms and data constructors for type level functions, fuzzing the core idea of the functions. Although the implementation of the data structures becomes much harder, the interface to the end-user does not change accordingly. We believe the benefits of enforcing structural properties of data structures in the their types is worth the extra effort.

In the end we think that the framework described in this document is a valid contribution to the list of storage solution for Haskell.

Acknowledgements

I would like to thank people.

Bibliography

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[BM98] Richard S. Bird and Lambert G. L. T. Meertens. Nested datatypes. In *MPC '98: Proceedings of the Mathematics of Program Construction*, pages 52–67, London, UK, 1998. Springer-Verlag.

[BS98] Douglas Barry and Torsten Stanienda. Solving the java object storage problem. *Computer*, 31:33–40, 1998.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cor09] Alberto Gmez Corona. Tcache: A transactional data cache with configurable persistence, 2009. `hackageDB: TCache`.

[GJ07] Neil Ghani and Patricia Johann. Initial algebra semantics is enough! In *Proceedings of Typed Lambda Calculus and Applications (TLCA), 2007*, number 4583 in Lecture Notes in Computer Science, pages 207–222, 2007.

[Goe09] John Goerzen. Hdbc: Haskell database connectivity, 2009. `hackageDB: HDBC`.

[HHMPJ05] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, to appear*, Jun 2005.

[Hin00] Ralf Hinze. Perfect trees and bit-reversal permutations. *J. Funct. Program.*, 10(3):305–317, 2000.

[HP06] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[Hu08] Wei Hu. personal communication, 2008.

[JGF96] Simon P. Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 1996. ACM.

[JJ99] Patrik Jansson and Johan Jeuring. Polytypic compact printing and parsing. In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, pages 273–287, London, UK, 1999. Springer-Verlag.

[Joh08] Patricia Johann. Foundations for structured programming with gadts. In *Conference record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–308, 2008.

[Kis09a] Oleg Kiselyov. Incremental multi-level input processing and collection enumeration, 2009. `http://okmij.org/ftp/Streams.html`.

[Kis09b] Oleg Kiselyov. iteratee: Iteratee-based i/o, 2009. `hackageDB: iteratee`.

[LK09] Don Stewart Lennart Kolmodin. binary: Binary serialisation for haskell values using lazy bytestrings, 2009. `hackageDB: binary`.

[LP03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[Mee92] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, September 1992.

[MFP91] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144. Springer-Verlag New York, Inc., 1991.

[MHJPJ08] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 11–20, New York, NY, USA, 2008. ACM.

[MP] Conor Mcbride and Ross Paterson. Idioms: applicative programming with effects.

[OHL06] Bruno C. D. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. volume 7 of *Trends in Functional Programming*. Intellect, 2006.

[O'R09] Neil Mitchell & Stefan O'Rear. derive: A program and library to derive instances for data types, 2009. `hackageDB: derive`.

[RHLJ08] Alexey Rodriguez, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. Technical Report UU-CS-2008-019, Department of Information and Computing Sciences, Utrecht University, 2008.

[RJJ$^+$08] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov, and Bruno C. d. S. Oliveira. Comparing libraries for generic programming in haskell. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 111–122, New York, NY, USA, 2008. ACM.

[SPJ93] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 106–116, New York, NY, USA, 1993. ACM.

[SPS98] Doaitse S. Swierstra, Pablo, and Joao Sariava. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.

[Ste09] Don Stewart. sqlite: Haskell binding to sqlite3, 2009. `hackageDB: sqlite`.

[SWP08] Sjaak Smetsers, Arjen Weelden, van, and Rinus Plasmeijer. Efficient and Type-Safe Generic Data Storage. In *Workshop on Generative Technologies*, Budapest, Hungary, April 5 2008. Electronic Notes in Theoretical Computer Science.

[VU98] Varmo Vene and Tarmo Uustalu. Functional programming with apomorphisms (corecursion). In *In 9th Nordic Workshop on Programming Theory*, 1998.