Comparing Approaches to Polytypic Programming

Joost Halenbeek

February 17, 1999

Contents

1 Introduction **3**

2 Categorical Datatypes in Haskell **5**

3 PolyP - a polytypic extension of Haskell **10**

3.1 Regular Datatypes and Functors 10

3.2 Type Inference 15

3.3 Semantics 16

3.3.1 Evidence Parameters 16

3.3.2 Partial Evaluation 17

3.4 Traversals 18

3.5 Shape Compression 22

3.5.1 HTML Documents 22

3.5.2 Compressing HTML Documents 24

3.5.3 Function `compress_shape` 26

3.5.4 Function `shape2bits` 26

3.5.5 Functions `pscM` and `fscM` 27

3.5.6 Function `putConstructor` 28

3.5.7 Function `decompress_shape` 28

3.5.8 Function `bits2shape` 29

3.5.9 Functions `psdM` and `fsdM` 29

3.5.10 Function `getConstructor` 30

3.5.11 Proof Sketch 31

3.6 Conclusions 32

4 Functorial ML **33**

4.1 Functors, Types and Terms 34

4.2 Two Isomorphisms 36

4.3 Mapping and Folding 40

4.4 Semantics 43

4.5 Traversals 43

4.6 Polytypic Definitions 46

4.7 Conclusions 47

5 Nested Datatypes **48**

5.1 What is a Nested Datatype? 48

5.2 Nested Datatypes as Higher-Order Functors 49

5.2.1 Higher-Order Functors 50

5.2.2 Catamorphisms 52

5.3 A Class of Nested Datatypes 53

5.3.1 A Polytypic `flatten` 57

5.3.2 A Polytypic `crush` Combinator 58

5.4 Higher-Order PolyP? 60

5.5 Conclusions 61

6 Conclusions **63**

A State Transformer Monads in Haskell **65**

B Implementing FML in Haskell **66**

B.1 The Functors 66

B.2 Mapping 67

B.3 Folding 68

B.4 Other Reduction Rules 68

B.5 Example: Cons-lists and Snoc-lists 69

Chapter 1

Introduction

Everyone who has had some experience in functional programming probably encountered the function map at some point. Given a function $f : A \to B$ the function map f takes a list of A and applies B to each element. The result is a list of B. In a suitable type system, e.g. the Hindley-Milner type system, the function map can be typed thus:

$$\mathsf{map} : \forall A, B.(A \to B) \to (List\ A \to List\ B)$$

The type parameters A and B are universally quantified, indicating that the algorithm for map is identical for different instantiations of A and B. We call map a *polymorphic* function and say it is *polymorphic in* A and B.

We can define a similar mapping function for other datatypes than lists, for example for trees. A map for trees applies a given function to all the leaves of a given tree. Many advantages are gained by having a single function for mapping over lists, trees and other datatypes. Such functions are often called *generic*, we shall call them *polytypic*. With a polytypic map, we are no longer forced to write a new mapping function for each new datatype that we encounter, and proofs with respect to this *polytypic* mapping function will be more general and hence more powerful. So we would like to have a mapping function with the following typing:

$$\mathsf{map} : \forall F.\forall A, B.(A \to B) \to (FA \to FB)$$

This typing is universally quantified with respect to a certain class of datatype constructors F. A polytypic mapping function that is typed as above does not describe a single algorithm, like a polymorphic function does, since the mapping algorithm for lists will obviously be different from the mapping algorithm for binary trees because lists have a different structure than trees. Just as the type constructor F ranges over a class of type constructors, a polytypic function describes not one, but an entire *class* of algorithms.

The following questions arise: What class of type constructors do we consider? What polytypic functions can we define with respect to this class? And how do we define these functions? In this thesis we will discuss three approaches to polytypy that answer these questions in different ways, although there may be more similarities than differences. For one thing, throughout this thesis we adopt the idea that datatypes are constructed with categorical functors. Category theory has had a major influence on thinking about polytypy and hence we start with a quick review of categorical datatype theory in Section 2. The theory is presented by implementing it in the functional programming language Haskell. In Section 3 we discuss PolyP ([JJ97, Jan97]), an implemented polytypic extension of Haskell that allows the user to define polytypic functions by induction on the structure of datatypes, or rather on the structure of their underlying functors. We show the practical use of PolyP by implementing functions for polytypic data compression. In Secton 4 we look at the language Functorial ML ([BJM96, JBM98]), which has not yet been implemented, but does approach polytypy in a more fundamental way than PolyP. In Section 5, we consider polytypy with respect to a different class of datatypes than the other two approaches: we describe nested datatypes with higher-order functors. We conclude with Section 6.

Chapter 2

Categorical Datatypes in Haskell

In this section, the reader is introduced to the categorical approach to datatype theory that originates with Hagino (see [Hag87]). The *Squiggol* community adopted this and showed its merits for program calculation and transformation (see [Mee86]).

We will present the categorical concepts that are necessary for understanding the subsequent sections by showing their implementation in Haskell. See also [MH95] for a similar implementation.

Categories. A category consists of a collection of objects and a collection of arrows. Each arrow has a source object and a target object. Two arrows can be composed to form a new arrow, if the first arrow has the same target object as the source object of the second arrow. This composition operation is associative. Furthermore, for each object `a` there is a so-called identity arrow with source and target `a`. An identity arrow is a neutral element of arrow composition.

The category that we use in Haskell is the category `Typ` that has types as objects and functions as arrows. Arrow composition in `Typ` is function composition `(.)` and the identity arrows in `Typ` are represented by the polymorphic function `id`. We call `Typ` our *base* category.

Functors. Functors provide mappings between categories. In Haskell, a functor from `Typ` to `Typ` is defined by a type constructor `f` of kind `* -> *`, for mapping objects to objects, together with a higher-order function `map`, for mapping arrows to arrows. Functors can be encapsulated as a constructor class as follows:

```
class Functor f where
   map :: (a -> b) -> (f a -> f b)
```

The arrow action of a functor must preserve identity arrows and distribute over arrow composition. For functors from `Typ` to `Typ`, this means that the following equations must hold:

```
map id      = id
map (f . g) = map f . map g
```

It is left to the Haskell user to make sure that only type constructors with a `map` that satisfies these equations are members of the class `Functor`.

A classic example of a functor is Haskell's type constructor for lists (`[]`), together with the well-known `map` for lists. We show how to describe this functor categorically when we discuss parameterized datatypes.

Bifunctors. The *product category* `Typ` $\times$ `Typ` consists of pairs of types and pairs of functions. We can define functors from `Typ` $\times$ `Typ` to the base category `Typ` in Haskell. These functors are called *bifunctors*. A (curried) bifunctor in Haskell is a type constructor of kind `* -> * -> *`, together with a function `map2`. Bifunctors can be represented by instances of the following constructor class:

```
class Bifunctor f where
   map2 :: (a -> c) -> (b -> d) -> (f a b -> f c d)
```

The following equations should hold:

```
map2 id id            = id
map2 (f . h) (g . k) = map f g . map h k
```

Examples of bifunctors are the categorical product and sum that are introduced below.

Products. Categorical *products* are provided in Haskell by the type constructor for pairs `(,)` and the projections `fst` and `snd`, together with a mapping `split` that satisfies the following equations:

```
fst . (split f g) = f
snd . (split f g) = g
```

Furthermore, the function `` f `split` g `` is the unique function that satisfies these equations.

A suitable `split` can be defined as follows:

```
split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)
```

The type constructor `(,)` can also be extended to a bifunctor by providing a suitable function `map2`:

```
(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (x,y) = (f x, g y)

instance Bifunctor (,) where
   map2 = (-*-)
```

Sums. Categorical *sums* can be defined in Haskell by means of the type constructor `Either`:

```
data Either a b = Left a | Right b
```

This type constructor implements categorical sums if there is a mapping `either` that satisfies the following equations:

```
(either f g) . Left  = f
(either f g) . Right = g
```

The function `either f g` is the unique function that satisfies these equations. We define `either` as follows:

```
either :: (a -> c) -> (b -> c) -> Either a b -> c
either f g (Left x)  = f x
either f g (Right y) = g y
```

The type constructor `Either` can be extended to a bifunctor by providing a suitable instantiation of `map2`:

```
(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d
(f -+- g) x = (Left . f) `either` (Right . g)

instance Bifunctor Either where
   map2 = (-+-)
```

Recursive Datatypes In category theory, a recursive datatype is modeled by an *initial algebra* with respect to an underlying functor.

Given a functor `f`, an initial `f`-algebra in Haskell consists of a type `t`, a function `in` and a mapping `cata` with the following typings:

```
in   :: f t -> t
cata :: (f a -> a) -> t -> a
```

The function `in` can be seen as a constructor for values of type `t` and `cata` is a combinator that can be used to define functions on `t`. Functions that are constructed with `cata` are called *catamorphisms*.

The following equation should hold in order for `(t, in, cata)` to be an initial `f`-algebra:

```
cata phi . in = phi . map (cata phi)
```

Furthermore, `cata phi` is the unique function that satisfies this equation.

In Haskell, the type `t` can be defined as the fix-point of the functor `f`. Consider the following recursive datatype of lists of integers:

```
data IntList = Nil | Cons Int IntList
```

We will define this datatype categorically as an initial algebra. First we define a functor `L` as follows:

```
data L a = Nil | Cons Int a

instance Functor L where
   map f Nil        = Nil
   map f (Cons x l) = Cons x (f l)
```

Now we can define an initial `L`-algebra `(Mu L, In, cata)` that defines lists of integers by providing the following definitions:

```
data Mu f = In (f (Mu f))

out :: Functor f => Mu f -> f (Mu f)
out (In x) = x

cata :: Functor f => (f a -> a) -> Mu f -> a
cata phi = phi . map (cata phi) . out
```

The type constructor `Mu` constructs the fix-point of a given functor `f`. The constructor function `In` replaces the function `in`. The function `out` is the inverse of `In` and is used as a destructor of datatypes that are defined with `Mu`. Note that we have defined `cata` generically for all Haskell functors. The definition of `cata` can be retrieved from the equation that characterizes initial algebras above.

Parameterized Datatypes. Recursive datatypes with parameters can also be modeled by initial algebras. This is done by parameterizing the underlying functors and making them into bifunctors.

The datatype of lists with parameter `a` can for instance be defined by means of the following bifunctor:

```
data L a b = Nil | Cons a b

instance Bifunctor L where
   map2 f g Nil        = Nil
   map2 f g (Cons x l) = Cons (f x) (g l)
```

We define a new fix-point constructor `Tau` that constructs parameterized datatypes from bifunctors and a catamorphism combinator for these datatypes as follows:

```
data Tau f a = In2 (f a (Tau f a))

out2 :: Bifunctor f => Tau f a -> f a (Tau f a)
out2 (In2 x) = x

cata2 :: Bifunctor f => (f a b -> b) -> Tau f a -> b
cata2 phi = phi . map2 id (cata2 phi) . out2
```

Type functors. When a datatype is constructed as an initial algebra of a bifunctor, the datatype itself, which is a type constructor, can be extended to a functor as follows:

```
instance Bifunctor f => Functor (Tau f) where
   map f = cata2 (In2 . map2 f id)
```

This functor is called the *type functor* of bifunctor `f`. The well-known function `map` on lists that applies a function to each entry of a given list is an example of the arrow mapping of a type functor.

Chapter 3

PolyP - a polytypic extension of Haskell

PolyP is a polytypic extension of the functional language Haskell that has been developed by Johan Jeuring and Patrik Jansson at Chalmers University of Technology in Sweden [JJ97, Jan97]. PolyP extends Haskell with a construct that allows the definition of polytypic functions by induction on the structure of user-defined datatypes.

In this section we will first discuss the notion of a functor in PolyP, then show how functors are used to represent datatypes. The reader is also introduced to a special construct that allows the user to define new polytypic functions.

After a brief discussion of type inference and semantics, we present an implementation of a polytypic shape-compression algorithm, that is polytypic with respect to all shapes of regular PolyP datatypes.

Finally we shall treat the question of how polytypic PolyP really is in the conclusions of this section.

3.1 Regular Datatypes and Functors

The polytypism of PolyP is restricted to a specific class of Haskell datatypes, so-called *regular* datatypes with one type argument. The Haskell declarations of these datatypes look like this:

```
data D a = C1 t11 ... t1m
         | ...
         | Cn tn1 ... tnm
```

where n ≥ 1 and m ≥ 0.

This declaration defines `D` to be a unary type constructor with constructors `C1`,…,`Cn`. The types `tij` can be instantiated with:

- the type parameter `a`
- the type `D a`, i.e. the application of `D` to type parameter `a`
- the application of another unary regular datatype to any of the types in this list `tij`
- a constant type such as `Int` or `String` or `Bool`

In order to define polytypic functions for these datatypes, we would like to be able to describe the structure of these datatypes inductively.

In Section 2 we have seen that initial algebras of bifunctors can be used to describe unary datatype constructors. PolyP uses an inductive class of bifunctors that can be used to describe any unary regular datatype. The bifunctors of PolyP are described by the following grammar:

$$F ::= G \texttt{ + } H \mid G \texttt{ * } H \mid \texttt{Empty} \mid \texttt{Par} \mid \texttt{Rec} \mid D \texttt{ @ } G \mid \texttt{Const } t$$

The nonterminals F, G and H describe bifunctors, D describes unary regular datatypes and t ordinary types.

The functors that are generated by this grammar are treated by PolyP as Haskell type synonyms in the following manner.

```
type (g + h) a b = Either (g a b) (h a b)
type (g * h) a b = (g a b, h a b)
type Empty   a b = ()
type Par     a b = a
type Rec     a b = b
type (d @ g) a b = d (g a b)
type Const t a b = t
```

The Haskell datatype `Either` implements disjoint sums. Its definition is given in Section 2.

We can see that:

- The terminal symbols `+` and `*` denote the binary categorical sum and product lifted to functors.

- `Empty` denotes the constant bifunctor that produces the terminal object.
- `Par` and `Rec` denote projection functors, sometimes referred to as Exl and Exr. These functors return the parameter and the recursive argument of a regular bifunctor respectively.
- `@` denotes the composition of a regular datatype with a bifunctor. Note that this yields again a functor, because regular datatype constructors are (unary) functors themselves.
- `Const` denotes a constructor for constant functors.

PolyP is able to generate a bifunctor for each unary regular datatype definition. Consider the following examples of the datatype of lists and rose trees, both regular datatypes.

```
data List a = Nil | Cons a (List a)
data Rose a = Fork a (List (Rose a))
```

The corresponding functors that PolyP calculates are:

```
FList = Empty + (Par * Rec)
FRose = Par * (List @ Rec)
```

Using the type synonyms above, the pointwise definitions of these functors are:

```
FList a b = Either () (a,b)
FRose a b = (a,List b)
```

Fixing the second parameter in these definitions yields datatypes that are isomorphic to the original definitions of `List` and `Rose`.

The action of a bifunctor on the level of types is given by the set of Haskell type synonyms. The action of a bifunctor on the level of functions can also be defined in Haskell, but we have to supply a separate mapping function for each bifunctor. PolyP allows us to describe the action of a functor on the level of functions with a single function `fmap` of the following type:

```
fmap :: Bifunctor f => (a -> c) -> (b -> d) -> f a b -> f c d
```

The context `Bifunctor f` indicates that the parameter `f` ranges over regular bifunctors.

We have seen that the class of regular bifunctors is defined inductively. This gives us the possibility to define `fmap` by induction over the structure of regular bifunctors. PolyP supplies the special `polytypic` construct to construct these inductive definitions. In general a polytypic definition looks like this:

$$\texttt{polytypic}\ \phi\ \texttt{::}\ \rho\ \texttt{=}\ \texttt{case}\ f\ \texttt{of}\ \{f_i \to e_i\}$$

where ϕ is the name of the value being defined, ρ its type and f a functor variable that may appear freely in ρ. The special `case` structure contains functor patterns f_i and expressions e_i.

A function defined in this fashion provides a template for constructing instances of polytypic functions given a specific PolyP functor. Informally, ϕ accepts a functor argument that is subsequently matched against the patterns in the body of the `case` to select the right expression. This way a single function that operates on all PolyP functors can be defined.

The function `fmap` can be defined with the `polytypic` construct as follows:

```
polytypic fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
  = \p r -> case f of
              g + h   -> fmap p r -+- fmap p r
              g * h   -> fmap p r -*- fmap p r
              Empty   -> id
              Par     -> p
              Rec     -> r
              d @ g   -> pmap (fmap p r)
              Const t -> id
```

The two auxiliary functions `-+-` and `-*-` implement functorial mapping on the Haskell `Either` data type and Haskell pairs respectively.

```
(-+-) :: (a -> c) -> (b -> d) -> Either a b -> Either c d
f -+- g = (Left . f) `either` (Right . g)

(-*-) :: (a -> c) -> (b -> d) -> (a,b) -> (c,d)
(f -*- g) (x,y) = (f x, g y)
```

The function `pmap`, that is used in defining `fmap` for functors composed with datatypes, implements the functorial mapping of a regular datatype. Its typing, which is polytypic with respect to the parameter d, is:

```
pmap :: Regular d => (a -> b) -> d a -> d b
```

The context `Regular d` indicates that the parameter `d` ranges over regular datatypes. The definition of `pmap` shall be given later in this section.

The exact relationship between a datatype and its underlying functor is made explicit by an isomorphism that is witnessed by the functions `inn` and `out`.

```
inn :: Regular d => (FunctorOf d) a (d a) -> d a
out :: Regular d => d a -> (FunctorOf d) a (d a)
```

The PolyP keyword `FunctorOf` provides access to the underlying functor of a regular datatype.

In categorical terms, given a regular datatype D, $\mathtt{inn}_D$ is the arrow that is associated with the initial algebra of the functor that represents D. In fact, $\mathtt{inn}_D$ is the categorical junc of the (uncurried) constructor functions of D. The following Haskell code implements the $\mathtt{inn}_{List}$ operation.

```
inn_List :: FList a (List a) -> List a
inn_List = Nil `either` uncurry Cons
```

Function `out` is the inverse of `inn` and is used in polytypic functions instead of pattern-matching on constructor functions. The implementation of `out` for a given datatype D, however, does use pattern-matching on the constructor functions of D. See for example the implementation of $\mathtt{out}_{List}$ below.

```
out_List :: List a -> FList a (List a)
out_List Nil         = Left ()
out_List (Cons x xs) = Right (x,xs)
```

Instantiations of `inn` and `out` for other datatypes can be implemented in a similar fashion. The PolyP compiler automatically generates instances of `inn` and `out` for all declared regular datatypes.

Regular datatype constructors can be extended to functors by providing the function `pmap`. We already used this function in the definition of `fmap`.

```
pmap :: Regular d => (a -> b) -> d a -> d b
pmap f = inn . fmap f (pmap f) . out
```

The resulting functor is the so-called *type* functor of the underlying bifunctor of the regular datatype.

In Section 2, this mapping of a type functor was defined as a catamorphism. The notion of a catamorphism is polytypic and can easily be defined in PolyP:

```
cata :: Regular d => ((FunctorOf d) a b -> b) -> d a -> b
cata phi = phi . fmap id (cata phi) . out
```

Now we can define `pmap` alternatively as a `cata` as follows:

```
pmap f = cata (inn . fmap f id)
```

This definition corresponds to the definition given earlier of a type functor.

Note that the defining equation for `cata` corresponds to the one discussed in Section 2.

3.2 Type Inference

The PolyP system is capable of inferring types for all expressions provided that all functions that are defined by means of the `polytypic` construct are explicitly typed. If this were not the case, higher-order unification, which is known to be undecidable, would be necessary to infer types.

The PolyP type system is based on an extension of the Hindley-Milner type system with qualified and higher-order polymorphic types, that is also used for type inference in an environment with constructor classes. (See Jones [Jon93])

For detailed information on the PolyP type system, see [Jan97]. Here we suffice by discussing the type checking of the `polytypic` construct.

Consider the inductive definition of a polytypic function ϕ:

$$\texttt{polytypic}\ \phi\ \texttt{::}\ \rho\ \texttt{=}\ \texttt{case}\ f\ \texttt{of}\ \{f_i \to e_i\}$$

Type checking makes sure that for each case-branch, the type that is obtained by instantiating ρ with the functor f_i is an instance of the type of e_i. The instantiation of ρ is obtained by interpreting f_i as a type synonym.

Consider for example a fragment of the the definition of `fmap`:

```
polytypic fmap :: (a -> c) -> (b -> d) -> f a b -> f c d
  = \p r -> case f of
              ...
              d @ g -> pmap (fmap p r)
              ...
```

First PolyP distributes the lambda abstraction `\p r ->` over the expressions in each case-branch. Considering the case-branch for functors `d @ g`, the following expression has to be type checked:

```
\p r -> pmap (fmap p r)
```

The type of this expression is:

```
(a -> b) -> (b -> d) -> d (g a b) -> d (g c d)
```

Substituting the functor `g + h` for `f` in the type signature of `fmap` yields the following type:

```
(a -> c) -> (b -> d) -> (d @ g) a b -> (d @ g) c d
```

Using the definition of `d @ g` by type synonym that was given before, PolyP rewrites this type to:

```
(a -> c) -> (b -> d) -> d (g a b) -> d (g c d)
```

This type is identical to the type of the expression in the case-branch. This means it is certainly an instance of that type and type checking succeeds for this part of the polytypic function.

3.3 Semantics

We have seen that PolyP extends Haskell expressions with built-in polytypic functions `inn` and `out` and with the means to define new polytypic functions with the `polytypic` construct. The semantics of PolyP is obtained by providing a translation of these new expressions to Haskell. We assume that the program is fully typed, i.e. that a most general typing for all expressions and subexpressions exists and is known.

3.3.1 Evidence Parameters

The translation takes place in two phases. First the program is translated to an intermediate language that contains no occurrences of the `polytypic` construct and no references to the classes `Regular` and `Bifunctor`. Instead, the translated program contains applications and abstractions of so-called *evidence parameters*.

Polytypic functions can be viewed as implicitly accepting an extra argument that contains information on how to instantiate the function. We call this extra information *evidence*. Evidence is either needed for functors or for regular datatypes. The evidence for a functor is just its symbolic representation as a PolyP functor. The evidence for a regular datatype D is a dictionary containing the instances of the functions `inn` and `out` with D and the functor representation of D. The use of evidence parameters is heavily based on the work of Jones on qualified types [Jon94].

We will illustrate the translation process by giving a translation of the function `fmap`. The definition of `fmap` that we gave earlier in terms of the `polytypic` construct is translated to the following code in the first phase.

$$
\begin{array}{l}
\texttt{fmap} = \lambda v.\ \texttt{\textbackslash p r -> case}\ v\ \texttt{of} \\
\qquad\qquad \begin{array}{lll}
g + h & \rightarrow & \texttt{fmap}_g\ \texttt{p r -+- fmap}_h\ \texttt{p r} \\
g * h & \rightarrow & \texttt{fmap}_g\ \texttt{p r -*- fmap}_h\ \texttt{p r} \\
\texttt{Empty} & \rightarrow & \texttt{id} \\
\texttt{Par} & \rightarrow & \texttt{p} \\
\texttt{Rec} & \rightarrow & \texttt{r} \\
d\ \texttt{@}\ g & \rightarrow & \texttt{pmap}_d\ (\texttt{fmap}_g\ \texttt{p r}) \\
\texttt{Const}\ t & \rightarrow & \texttt{id}
\end{array}
\end{array}
$$

Notice that the `polytypic` construct has been replaced by evidence abstraction and a special `case` over evidence. Applications of polytypic functions have an extra evidence parameter that is indicated by a subscript. Consider for example the expression

$$\texttt{pmap}_d\ (\texttt{fmap}_g\ \texttt{p r})$$

It contains an application of polytypic function `pmap` to evidence parameter d that contains evidence for some regular datatype. Similarly, `fmap` is applied to evidence parameter g that contains evidence for a regular bifunctor.

3.3.2 Partial Evaluation

The second phase uses partial evaluation of evidence parameters to calculate the required instances of polytypic functions. Consider for example application of polytypic mapping to a list structure. The required instance of `pmap` will be $\texttt{pmap}_{List}$, i.e. `pmap` applied to evidence for the datatype *List*.

$$\texttt{pmap}_{List} = \texttt{\textbackslash f -> inn}_{List}\ \texttt{.}\ \texttt{fmap}_{FList}\ \texttt{.}\ \texttt{out}_{List}$$

where FList is the functor that underlies the List datatype.

Instances of inn and out for the datatype List and an instance of fmap for the functor FList are required to translate the above instance of pmap. We have already shown the required instances of inn and out in a previous section. The PolyP system is able to generate instances of these functions for any regular datatype automatically from the datatype declaration.

The computation of the instance $\texttt{fmap}_{FList}$ involves a number of rewrite steps that inductively break down the functor *FList* using the definition of fmap.

Here is the reduction. The result is an expression containing no evidence parameters.

$$
\begin{array}{ll}
 & \texttt{fmap}_{FList} \\
\rightarrow & \texttt{fmap}_{Empty+(Par*Rec)} \\
\rightarrow & \texttt{\\p r -> fmap}_{Empty}\texttt{ p r -+- fmap}_{Par*Rec}\texttt{ p r} \\
\rightarrow & \texttt{\\p r -> id -+- (\\p r -> fmap}_{Par}\texttt{ p r -*- fmap}_{Rec}\texttt{ p r) p r} \\
\rightarrow & \texttt{\\p r -> id -+- (p -*- r)}
\end{array}
$$

The translation of $\texttt{fmap}_{FList}$ is complete when all references to it are replaced by a new identifier, say fmap_FList. Similarly all references to $\texttt{pmap}_{List}$ are replaced by a new identifier, say pmap_List. The result of the translation is the following code.

```
fmap_FList = \p r -> id -+- (p -*- r)
pmap_List = \f -> inn_List . fmap_FList f pmap_List . out_List
```

We already saw implementations of inn_List and out_List in the beginning of this chapter.

3.4 Traversals

Informally, a traversal visits each node of a given structured value and performs some action on that node. Traversals can be defined as monadic maps. We assume the reader is familiar with monads and we use monads as declared with the constructor class Monad in the Haskell prelude. The monad's bind operation is denoted by >>=, the return operation is denoted by return, x >> y is defined as x >>= _ -> y and monadic composition is denoted by @@.

The function mapM from the Haskell prelude defines monadic maps for lists:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

```
mapM f []     = return []
mapM f (x:xs) = f x       >>= \y  ->
                mapM f xs >>= \ys ->
                return (y:ys)
```

This monadic map traverses a list from head to tail and applies a function of type `a -> m b` to each entry. The resulting computations of type `m b` are combined with the bind operation `>>=`. This type of traversal is called *left traversal*. A *right traversal* on a list traverses the list from tail to head. Here is how right traversal on lists is implemented in Haskell:

```
mapMr :: Monad m => (a -> m b) -> [a] -> m [b]
mapMr f []     = return []
mapMr f (x:xs) = mapMr f xs >>= \ys ->
                 f x        >>= \y  ->
                 return (y:ys)
```

Notice that the only difference with left traversal is the order in which the monadic computations `f x` and `mapMr f xs` are performed.

PolyP's polytypism can be combined with Haskell's monadic programming facilities to define traversals for all unary regular datatypes. Polytypic left traversal is defined by the function `pmapM` thus:

```
pmapM :: (Regular d, Monad m) => (a -> m b) -> d a -> m (d b)
pmapM fM = innM @@ fmapM fM (pmapM fM) @@ outM

polytypic fmapM :: Monad m => (a -> m c) -> (b -> m d) ->
                              f a b -> m (f c d)
  = \p r -> case f of
       g + h     ->  summapM  (fmapM p r) (fmapM p r)
       g * h     ->  prodmapM (fmapM p r) (fmapM p r)
       Empty     ->  return
       Par       ->  p
       Rec       ->  r
       d @ g     ->  pmapM (fmapM p r)
       Const t   ->  return
```

The functions `innM` and `outM` are monadic versions of `inn` and `out`. They are defined thus:

```
innM :: (Regular d, Monad m) => (FunctorOf d) a (d a) -> m (d a)
```

```
innM = return . inn

outM :: (Regular d, Monad m) => d a -> m ((FunctorOf d) a (d a))
outM = return . out
```

The function `pmapM` is defined in terms of the function `fmapM`, which defines left traversal for bifunctors. Left traversal for the bifunctor `+` is defined by the function `summapM`:

```
summapM :: Monad m => (a -> m c) -> (b -> m d) ->
                      Either a b -> m (Either c d)
summapM f g = (mapm Left . f) `either` (mapm Right . g)
```

The function `mapm` defines the map of a monad (monads are also functors). It is defined thus:

```
mapm :: Monad m => (a -> b) -> (m a -> m b)
mapm f x = x >>= \y -> return (f y)
```

Left traversal for the bifunctor `*` is defined by the function `prodmapM`:

```
prodmapM :: Monad m => (a -> m c) -> (b -> m d) ->
                       (a,b) -> m (c,d)
prodmapM  f g p = f (fst p) >>= \x ->
                  g (snd p) >>= \y ->
                  return (x,y)
```

We can define functions `pmapMr` and `fmapMr` for polytypic right traversal by substituting the function `prodmapM` by the function `prodmapMr`, which is defined thus:

```
prodmapMr :: Monad m => (a -> m c) -> (b -> m d) ->
                        (a,b) -> m (c,d)
prodmapMr  f g p = g (snd p) >>= \y ->
                   f (fst p) >>= \x ->
                   return (x,y)
```

This function traverses a pair from second to first element, whereas `prodmapM` traverses a pair from first to second element. This means that, compared with left traversal, right traversal visits the data of a structure in reverse order.

Shape and data decomposition. Traversals are very useful polytypic building blocks that can be used to define a variety of polytypic functions. A possible application is the decomposition of a value of a regular datatype into its shape and its contents. The shape of a value `x` of type `d a` is a value of type `d ()` that represents the structure of `x`. This shape can be obtained by applying the function `pmap (const ())` to `x`. We could define a polytypic `flatten` for the extraction of the contents of a value and define the decomposition of values as `` pmap (const()) `split` flatten ``, but then the structure of `x` would have to be traversed twice. Instead, we define a function `separate` that computes the shape and contents of a value by traversing that value only once.

We use the monad `State [a]` for decomposition of values of type `d a`. This is a state transformer monad where the state is a list with values of type `a`. The definition of `State` in Haskell can be found in Appendix A.

The function `separate` decomposes a value into its shape and contents by performing a right traversal and performing the action defined by the function `put` at each datum that the value contains. The function `put` prepends the datum to the state and replaces the datum by `()`. Here is its definition:

```
separate :: Regular d => d a -> (d (), [a])
separate x = pmapMr put  >>= \x ->
             fetchState  >>= \s ->
             return (x,s) `startingWith` []

put :: a -> State [a] ()
put x = ST (\s->((),x:s))
```

Using the same monad, we can define a function `combine`, which is a partial inverse of `separate`. Given a shape of type `d ()` and a list of data of type `a`, the function `combine` fills the 'holes' in the shape with the data by means of a left traversal with the function `get`. The result is a value of type `d a`, provided that there is enough data to fill the shape.

```
combine :: regular d => (d (),[a]) -> d a
combine (x,ps) = pmapM get x `startingWith` ps

get :: () -> State [a] a
get () = ST (\l -> case l of
                 []     -> error "not enough data to fill shape"
                 (x:xs) -> (x,xs))
```

Note that `separate` is defined as a left traversal and `combine` is defined as a right traversal. In fact they are partial inverses.

3.5 Shape Compression

The compression algorithms that are used by conventional compression utilities, such as Unix compress, usually have no knowledge of structure in the data they receive as input. If one knows beforehand that the input is for example a Pascal or Java program, or an HTML document, the results of compression is usually much more compact.

The compression strategy is as follows: parse the input into a structured value, separate data from shape (i.e. structure), compress the shape to a string with a specific shape compression algorithm and compress this string together with the data with a conventional compression algorithm. Decompression proceeds in the opposite direction: decompress with the conventional algorithm to obtain the data and the compressed shape, decompress the shape, combine shape and data into a structured value and unparse this value.

These strategies are polytypic by nature. In this section we implement polytypic shape compression and decompression algorithms in PolyP. The algorithms are polytypic with respect to all of PolyP's regular datatypes. We use monadic maps, also known as *traversals* to implement functions for separation and combination of data and shape. We show how to implement functions for shape compression and shape decompression by a slight adaptation of the definitions for traversals.

3.5.1 HTML Documents

As an example throughout this section, we consider the compression of HTML documents. A simplified datatype of HTML documents can be defined as follows:

```
data Html = Text String
          | Heading Int Html
          | Image Url
          | List [Html]
          | Link Url Html
          | Break
          | HRule

type Url = String
```

This datatype is regular but it is not parameterized. This implies that PolyP does not consider it to be regular and is unable to instantiate polytypic functions for it. PolyP only supports polytypism for unary regular datatypes.

We can define a unary datatype constructor `Html2` that models `Html` by introducing a dummy parameter. The new datatype is regular in PolyP.

```
data Html2 a = Text String
             | Heading Int (Html2 a)
             ...

type Html = Html2 ()
```

The problem with this approach is that we cannot obtain the data of a value by flattening, because the data is not stored at parameter positions. Instead, it is stored at positions marked by type constants in the declaration, e.g. `String` and `Int`.

One way of extracting the 'hidden' data is to use Haskell's overloaded `show` function to represent all data as strings. During decompression `read` can be used to recover the original data. The problem of this method is that the compression results will depend on the implementation of `show`. Moreover, we prefer not to lose type information during extraction. This makes it possible to use compression algorithms for specific types.

An approach that suits us better, is to parameterize all types that appear as type constants in the declaration of `Html`:

```
data Html3 a b c = Text a
                 | Heading b (Html3 a b c)
                 | Image c
                 | List [Html3 a b c]
                 | Link c (Html3 a b c)
                 | Break
                 | HRule

type Html = Html3 String Int String
```

The new datatype constructor `Html3` has different parameters for the types of text, headings and url's. The type of HTML documents is defined by instantiating `Html3` with the types `String` (for text), `Int` (for the level of headings) and `String` (for url's).

PolyP doesn't support polytypism for this datatype because it has more than one parameter. We can solve this by modeling the three parameters by a single parameter as follows:

```
data Html4 a = Text a
             | Heading a (Html4 a)
             | Image a
             | List [Html4 a]
```

```
              | Link a (Html4 a)
              | Break
              | HRule

data ParType = TextPar String
             | HeadingPar Int
             | UrlPar String

type Html = Html4 ParType
```

The single parameter of `Html4` is instantiated with the type `ParType` to declare the type of HTML documents. `ParType` is a disjoint sum of the three types of data (text, heading or url) that an HTML document can contain.

The datatype `Html` declared in terms of `Html4` as above is regular in PolyP. We will use this datatype as an example to illustrate shape (de)compression.

3.5.2 Compressing HTML Documents

Consider the following HTML document:

> My Home Page
>
> ---
>
> Links to my friends:
> - Martijn
> - Jantje

This document can be represented by the datatype `Html4` as follows:

```
my_html :: Html4 ParType
my_html = List [ Heading (HeadingPar 1)
                         (Text (TextPar "My Home Page"))
               , HRule
               , Break
               , Heading (HeadingPar 2)
                         (Text (TextPar "Links to my friends:"))
               , List [ Link (UrlPar "http://...martijn...")
                             (Text (TextPar "Martijn"))
                      , Link (UrlPar "http://...michiel...")
                             (Text (TextPar "Jantje"))
                      ]
               ]
```

To compress `my_html` using the strategy described above, its shape and data must be separated. The data can be obtained by flattening, it is a list of values of type `ParType`:

```
[ HeadingPar 1
, TextPar "My Home Page"
, HeadingPar 2
, TextPar "Links to my friends:"
, UrlPar "http://...martijn..."
, TextPar "Martijn"
, UrlPar "http://..jantje..."
, TextPar "Jantje"
]
```

The shape of `my_html` is obtained by replacing each datum by `()`, i.e. by leaving out the data. Since all data is declared as a parameter, the shape consists of just constructors.

The first step in shape compression is flattening the shape to a list of constructors. Here is a representation of the flattened shape of `my_html`:

```
[ List, Cons, Heading, Text
      , Cons, HRule
      , Cons, Break
      , Cons, Heading, Text
      , Cons, List, Cons, Link, Text
                  , Cons, Link, Text
                  , Nil
      , Nil
]
```

The list is formatted such that the represented structure is still visible.

Since all datatypes in Haskell are sum-of-product datatypes, all datatypes in PolyP are sum-of-product datatypes too. The constructors of any datatype in PolyP can therefore be coded as natural numbers indicating which injection into the sum each constructor represents. These numbers are easily retrieved by inspecting the `data` declaration of a given datatype. We call such a number the *index* of a constructor. The shape compression algorithm represents the indices of the constructors of a given datatype by bit strings, whose length depends on the number of constructors of the datatype. The flattening of the shape and the encoding of constructors in bit strings is merged in the actual implementation of the algorithm.

The list of bits is chopped up into bytes. Each byte represents an ASCII character, so the shape can be represented as a string, which is final result of shape compression.

Example. The constructors `Nil` and `Cons` of type constructor `[]` are assigned the indices `0` and `1` respectively. These indices can be represented by a single bit. The constructors `Text`, `Heading`, ..., `HRule` of `Html4` have indices `0`, `1`, ..., `6` respectively. These indices can be represented by bit strings of length 3, i.e. `000` represents `0`, `001` represents `1` and `110` represents `6`.

The shape of `my_html` is represented by the following list of bits:

011 1 001 000 1 110 1 011 1 001 000 1 011 1 100 000 1 100 000 0 0

The list is formatted such that each group of 3 bits represents a constructor of `Html4` and each single bit represents a constructor of `[]`.

The shape compression algorithm encodes the list of bits above by the following string of 6 characters: `"r:E\224\192\DEL"`. Given enough type information, the decompression algorithm can recover the original shape of `my_html` from this string.

The functions for shape compression and decompression are slight variations of the left and right traversals we discussed in Section 3.4.

3.5.3 Function `compress_shape`

The main function for shape compression is `compress_shape`. It compresses a given shape of any regular datatype to a list of ASCII characters, i.e. a string.

```
compress_shape :: Regular d => d () -> String
compress_shape = encode_in_ASCII . shape2bits
```

The compression is performed in two steps. First the shape is flattened and represented by bits by the function `shape2binary`. The bits are encoded into ASCII by `encode_in_ASCII`. We proceed with a discussion of `shape2binary`. The implementation of the function `encode_in_ASCII` is omitted.

3.5.4 Function `shape2bits`

The function `shape2bits` compresses a shape to a list of bits as was illustrated by an example above.

```
shape2bits :: Regular d => d () -> [Bit]
shape2bits x = (pscM idM x >> fetchState) `startingWith` []
```

The monadic function `pscM idM x` is a state transformer. The monad we use is `State [Bit]`, which is a monad that models a global state that contains a list of bits. See Appendix A for the Haskell definition of this monad.

The state transformer `pscM idM x` is executed with the function `startingWith` with the empty list as initial state. The required list of bits is the final state, which is obtained by `fetchState`.

The function `pscM idM` does the actual work. It performs a right traversal on its argument and each time a constructor is encountered, the bit string corresponding to its index is concatenated to the list in the state.

3.5.5 Functions `pscM` and `fscM`

These functions implement a kind of right traversal of a given tree-shape, just like the functions `pmapMr` and `fmapMr` from Section 3.4. However, side-effects are not just collected from the leaves (i.e. the data) of the tree, but from the nodes (i.e. the constructors) as well.

```
pscM :: Regular d => (a -> State [Bit] b) -> d a -> State [Bit] (d b)
pscM fM = putConstructor @@ innM @@ fscM fM (pscM fM) @@ outM

polytypic fscM :: (a -> State [Bit] c) -> (b -> State [Bit] d) ->
                  f a b -> State [Bit] (f c d)
  = \p r -> case f of
       g + h   -> summapM   (fscM p r) (fscM p r)
       g * h   -> prodmapMr (fscM p r) (fscM p r)
       Empty   -> idM
       Par     -> p
       Rec     -> r
       d @ g   -> pscM (fscM p r)
       Const t -> idM
```

These definitions are very similar to those of `pmapMr` and `fmapMr`, but here we use a specific monad: `State [Bit]`. This monad defines state transformers. Its definition in Haskell can be found in Appendix A.

The recursive body of the definition of `pscM` is different from the body of `pmapMr`: it is composed with the function `putConstructor`. Computationally this means that `putConstructor` is applied at every node of the shape during the traversal.

3.5.6 Function putConstructor

The function putConstructor accepts a value, say x, constructed by some regular datatype d. Since d is a regular datatype, x is constructed using some constructor function, say C. The function constructor2Int, which is in PolyP's library, retrieves C and returns the number that represents it. The function int2binary' n encodes this number into a list of bits of length n, which is prepended to the state with the function updateState.

The bits are appended to the state by means of the function updateState.

```
putConstructor :: Regular d => d a -> State [Bit] (d a)
putConstructor x
  = let
       n = length (int2binary (no_constructors x - 1))
     in
       updateState (\l -> int2binary' n (constructor2Int x) ++ l) >>
       return undefined
```

The required number n is the length of the bit string that represents the highest constructor index. This length is defined in the let-construction by means of the function no_constructors. This function returns the number of different constructors the datatype of its argument has. Note that the constructors of a given datatype are represented by bit strings of a fixed length to make decompression possible.

We have used two functions that convert integers to bit strings. The function int2binary returns a bit string without any zeros at its left end. The function int2binary' accepts an extra parameter that indicates the length of the bit string it returns. The resulting bit string of the latter function may have zeros at its left end.

3.5.7 Function decompress_shape

Decompression is performed by the function decompress_shape. It is the left inverse of compress_shape. The function decode_from_ASCII converts the compressed shape (a string) to a list of bits. The compressed shape is recreated by the function bits2shape.

```
decompress_shape :: Regular d => String -> d ()
decompress_shape = bits2shape . decode_from_ASCII
```

3.5.8 Function `bits2shape`

The function `bits2shape` is the left inverse of `shape2bits`. It recovers a shape from the list of bits that represents it. The function is defined using the monadic function `psdM`.

```
bits2shape :: Regular d => [Bit] -> d ()
bits2shape l = (psdM idM undefined) `startingWith` l
```

The shape is being built by a state transformer that is defined as a monad. The state transformer `psdM idM undefined` can be seen as a polytypic parser that parses a list of bits to a shape of a regular datatype.

3.5.9 Functions `psdM` and `fsdM`

The functions `psdM` and `fsdM` are almost identical to the functions `pscM` and `fscM` that were used for compression, but for some small, yet essential differences. Firstly, the functions `psdM` and `fsdM` perform a kind of left traversal, contrary to the right traversal of `pscM` `fscM`. Secondly, the function `putConstructor` is replaced by `getConstructor`. Thirdly, in `psdM` the function `getConstructor` is applied at each node, *before* traversing its children rather than afterwards, as was the case with `putConstructor` in `pscM`. The latter fact means that `psdM` employs a depth-first strategy for traversal, whilst `pscM`'s strategy is breadth-first.

Recall that the function `putConstructor` is a monadic function whose side-effect is adding a bit string, that represents a constructor, to the global state. Similarly, the function `getConstructor` is a monadic function whose side-effect is reading a bit string from the state. In addition, the represented constructor is retrieved from the bits and returned as function result. We shall illustrate the process of shape decompression with our example.

Example: decompression. Below is a table that shows how the shape that is under construction and the state evolve during five consecutive times that `getConstructor` is applied.

```
undefined                                     011 1 001 000 1 110 ...
List undefined                                    1 001 000 1 110 ...
List (Cons undefined undefined)                     001 000 1 110 ...
List (Cons (Heading undefined) undefined)               000 1 110 ...
List (Cons (Heading (Text ())) undefined)                   1 110 ...
...
```

At first the shape is completely undefined except for its type (`Html4 ()`) and the state contains the bit string that represents the compressed shape. Before any traversal is done, `getConstructor` reads the first three bits from the state and identifies these as the constructor `List`. This constructor is returned with all its children undefined (`List undefined`) and decompression proceeds by traversing this structure in a depth-first manner from left-to-right, applying `getConstructor` to each sub-structure before continuing traversal.

Here are the definitions of `psdM` and `fsdM`:

```
psdM :: Regular d => (a -> State [Bit] b) -> d a -> State [Bit] (d b)
psdM fM = innM @@ fsdM fM (psdM fM) @@ outM @@ getConstructor

polytypic fsdM :: (a -> State [Bit] c) -> (b -> State [Bit] d) ->
                   f a b -> State [Bit] (f c d)
  = \p r -> case f of
       g + h   -> summapM  (fsdM p r) (fsdM p r)
       g * h   -> prodmapM (fsdM p r) (fsdM p r)
       Empty   -> idM
       Par     -> p
       Rec     -> r
       d @ g   -> psdM (fsdM p r)
       Const t -> idM
```

Note that the code strongly resembles the code for polytypic left traversal (functions `pmapMl` and `fmapMl`), see Section 3.4.

3.5.10 Function `getConstructor`

As mentioned above, the function `getConstructor` reads a number of bits from the state and returns the constructor that is represented by these bits.

```
getConstructor :: Regular d => d a -> State [Bit] (d a)
getConstructor x
  = let
       n = length (int2binary (no_constructors x - 1))
     in
       updateState (drop n) >>= \l ->
   return (int2Constructor (binary2int (take n l)))
```

The argument of `getConstructor` is in practice always the value `undefined`. What's important is the type, `d a`, of the argument. This type is used to calculate the number of bits `n` that have been used to encode a constructor of `d`.

The body of getConstructor converts the first n bits of the state to a constructor by consecutively applying binary2int and int2Constructor.

3.5.11 Proof Sketch

In order to prove that the shape decompression algorithm recovers the compressed shape, it is required to prove that:

```
decompress_shape . compress_shape = id
```

After expanding the definitions of decompress_shape and compress_shape, we can see that it suffices to prove the following two equations:

```
decode_from_ASCII . encode_in_ASCII = id
bits2shape . shape2bits = id
```

The second of these equations is the interesting one. Expanding definitions yields:

```
  \x -> (psdM idM undefined) `startingWith`
        ((pscM idM x >> fetchState) `startingWith` [])
=
  id
```

The construction on the left hand side of the equation applies the monadic function for compression (pscM idM) to x, it discards the result and returns the final state instead. This state is then used as the initial state of the state transformer that implements decompression (psdM idM undefined). Instead of the complicated combination of startingWith, fetchState and >>, we can use a single monadic operation >> that passes the state from one state transformer to the other and discards the result, to achieve the same behaviour. This means that the equation above is equivalent to the following one:

```
  \x -> (pscM idM x >> psdM idM undefined) `startingWith` []
=
  id
```

Once the decompression function psdM idM is instantiated with a concrete regular datatype, it is independent of its argument. It only depends on the initial state that contains the compressed shape. This means that we might as well pass the result from compression, i.e. the result of the state transformer pscM idM x, to the state transformer for decompression. This yields the following equation:

```
  \x -> (pscM idM x >>= \y -> psdM idM y) `startingWith` []
=
  id
```

Replacing the monadic bind with monadic composition yields:

```
\x -> (pscM idM @@ psdM idM) x `startingWith` [] = id
```

We can easily prove that

```
\x -> idM x `startingWith` [] = id
```

So it suffices to prove that

```
pscM idM @@ psdM idM = idM
```

Inspection of the definitions of `pscM` and `psdM` tells us that they are defined symetrically and that it should be enough to prove that `getConstructor` is the left inverse of `putConstructor`. The rest of the proof uses induction on the structure of functors and fixpoint induction.

3.6 Conclusions

We have shown how PolyP extends the functional programming language Haskell with a polytypic construct that allows the user to define polytypic functions by induction on the structure of functors that underly the datatypes. PolyP has been implemented as a compiler that translates polytypic programs to plain Haskell. The powerful combination of polytypy with the general monads of Haskell can be used to describe polytypic traversals with. These traversals were the inspiration for the PolyP implementations of polytypic compression and decompression algorithms for PolyP datatypes.

Currently, PolyP is only able to describe unary regular datatypes with functors. Future versions of PolyP should be able to cope with mutual recursive datatypes, function spaces, and datatypes with multiple type arguments, which we know can be described by functors of higher arities.

Chapter 4

Functorial ML

Functorial ML, hereafter abbreviated as FML, is an extension of the Hindley-Milner type system, hereafter abbreviated as HM, with functors. HM is at the core of most functional programming languages, like ML and Haskell. The HM type system allows for a restricted form of polymorphism called *let*-polymorphism. This form of polymorphism has the nice property that a type inference algorithm exists and is easily implemented. FML extends HM with functors and offers the possibility to parameterize types over these functors. This leads to a notion called *functorial polymorphism* which is the FML term for of polytypy. Together with the type system comes a type inference algorithm and an algorithm for the reduction of the polytypic operations map and fold (for catamorphisms) that can be extended to incorporate the reduction of other polytypic operations like traversals.

Most approaches implement polytypic map functions using an algorithm that in some way depends on the functor parameter. Programming languages like Charity [CF92] and PolyP [JJ97, Jan97] generate the algorithms for specific functors automatically. In Haskell, the user has to supply the algorithms explicitly by instantiating a *constructor class* [Jon93]. The algorithm for mapping in FML however is *uniform*, meaning it is a single algorithm that works for any FML functor. This uniformity is possible because of the use of special combinators that help to find the data inside an arbitrary structure that is built with FML functors.

The syntax of FML's functors, types and terms and a subset of the combinators is discussed in Section 4.1. In Section 4.2 the special combinators are discussed and in the Section 4.3 we discuss polytypic mapping and folding. After a brief look at semantics in Section 4.4, polytypic traversals are the topic of Section 4.5. Before concluding with Section 4.7, we briefly look at an extension of FML that supports polytypic definitions in Section 4.6.

4.1 Functors, Types and Terms

Functors. As already mentioned, the FML type system extends HM with functors. FML functors are constructed by the following grammar:

$$F, G \ ::= \ X \mid C \mid \Pi_i^m \mid F\langle\overline{G}\rangle^n \mid \mu^m F$$

A functor Π_i^m denotes an m-ary projection functor that returns the i-th type of the m types it is applied to. A functor $F\langle\overline{G}\rangle^n$ denotes an n-ary functor that is composed from an m-ary functor F and m functors $\overline{G}$ of arity n. The notation $\overline{G}$ means a vector of functors $G_0, G_1, \ldots, G_{m-1}$. A functor $\mu^m F$ denotes the m-ary type functor of an $m{+}1$-ary functor F, i.e. the fix-point of the functor that is obtained by fixing the first m parameters of F. In addition to these functors, which are the basic functors of FML, functors include functor variables X and functor constants C. Currently the functors $+{:}\,2$, $\times{:}\,2$ and $\mathbf{1}{:}\,0$ for disjoint sums, products and the unit type are supported. Note that the unit type is defined as a nullary functor $\mathbf{1} : 0$, not as a type. We shall denote the unit type, which is obtained by functor application of the unit functor to zero types, with $\mathbf{1}$ as well.

Types and Type Schema. FML types are HM types extended with the possibility to apply an m-ary functor to a vector of m types.

$$\tau \ ::= \ X \mid F(\overline{\tau}) \mid \tau_1 \to \tau_2$$

Functor application corresponds categorically to the object mapping of the functor. This means that whenever F is a functor of arity m and $t_1, .., t_m$ are types, $F(t_1, .., t_m)$ is a type too.

FML types can be quantified by type variables and by functor variables to obtain *type schema*, which are described by the following grammar:

$$\sigma \ ::= \ \tau \mid \forall X{:}T.\sigma \mid \forall X{:}m$$

The quantification of functors is new and is used to type functorial polymorphic terms. For example, the type of the map combinator for functors of arity 1 is:

$$\mathsf{map}_1 : \forall F{:}1.\ \forall X, Y.\ (X \to Y) \to FX \to FY$$

Terms. The terms of FML are like those of HM, i.e. λ-terms and a let-construct, but there is no fix-point combinator. This means that FML does not facilitate general recursion, but the FML reduction system can be proved to be strongly normalizing. Instead of general recursion FML supports catamorphisms for arbitrary functors by means of the combinator fold. Here is the grammar for FML terms:

$$t \ ::= \ x \mid c \mid \lambda x.t \mid t_1 \ t_2 \mid \mathsf{let} \ x = t_1 \ \mathsf{in} \ t_2$$

The terms are extended with a set of combinators c [1] , including the combinators map and fold that were mentioned above, for functors of any arity. More on mapping and folding in Section 4.3. Other combinators include constructors and destructors with respect to the FML functors. For the functors that are added as constants, we have the following combinators: pair for constructing pairs, pair projections π_0 and π_1, injections in_0 and in_1 for constructing sum values, case for case analysis of a sum value and un, denoting the canonical term of the unit type. Types that are constructed with a functor of the form $\mu^m F$ (where F is an m-ary functor) form an initial algebra with the combinators wrap_m for construction and $fold_m$ for destruction of values.

Here are the types that are assigned to the combinators we have seen so far:

$$\begin{array}{rcl}
\mathsf{pair} & : & \forall X, Y : T.X \to Y \to X \times Y \\
\pi_0 & : & \forall X, Y : T.X \times Y \to X \\
\pi_1 & : & \forall X, Y : T.X \times Y \to Y \\
\mathsf{in}_0 & : & \forall X, Y : T.X \to X + Y \\
\mathsf{in}_1 & : & \forall X, Y : T.Y \to X + Y \\
\mathsf{case} & : & \forall X, Y, Z : T.(X \to Z) \to (Y \to Z) \to X + Y \to Z \\
\mathsf{un} & : & \mathbf{1} \\
\mathsf{wrap}_m & : & \forall F : m{+}1.\forall X_{i \in m} : T.F(\overline{X}, \mu^m F(\overline{X})) \to \mu^m F(\overline{X}) \\
\mathsf{fold}_m & : & \forall F : m{+}1.\forall X_{i \in m}, Y : T.(F(\overline{X}, Y) \to Y) \to \mathsf{T}^{(m)} F(\overline{X}) \to Y \\
\mathsf{map}_m & : & \forall F : m.\forall X_{i \in m}, Y_{i \in m} : T.(X_i \to Y_i) \to_{i \in m} F(\overline{X}) \to F(\overline{Y})
\end{array}$$

The notation $\forall X_{i \in m}$ is short for $\forall X_0, X_1, \ldots, X_{m-1}$. The notation $\overline{X}$ is used instead of $X_{i \in m}$ whenever it is clear which types are referred to. Note that the typings for wrap, fold and map denote the types of an entire family of combinators for $m \geq 0$.

The combinators map and fold are discussed in more detail in Section 4.3. Combinators for construction and destruction with respect to composite functors and projection functors are the subject of the next section.

[1] In this thesis we consider the set of FML combinators that was used in [BJM96], but we have renamed them a la [JBM98].

Type Inference. Milner's type inference algorithm W, which infers principal typings for HM terms, can be modified to calculate most general typings for FML terms. The most important modification is that not only types, but also functors have to be unified. For a description of the FML type inference algorithm, see [BJM96].

4.2 Two Isomorphisms

Consider mapping over a value of type $F(GA)$. This value is constructed from functors F, G and type A. There are two ways of performing the mapping: we can either use the map from the functor $F \circ G$, or we can use the map from the functor F. Which map is to be used can often be inferred from the type of the function that is being mapped with.

For instance: given a functor T that constructs trees and a functor L that constructs lists, consider mapping the function $succ : \mathbb{N} \to \mathbb{N}$ (producing the successor of a natural number) over a tree of lists of naturals, i.e. a value of type $T(L\mathbb{N})$. From the type of $succ$ we can infer that the functor that should be used for mapping is $T \circ L$, and not T. This is because $succ$ can only be applied to naturals. If $succ$ is mapped using the map for T, then it would have to be applied to lists of naturals, which is impossible.

This is the fully explicit type of the correct instantiation of the application of map:

$$\mathsf{map}_1[T \circ L][\mathbb{N}][\mathbb{N}]\ succ : T(L\mathbb{N}) \to T(L\mathbb{N})$$

When the argument of map is polymorphic, ambiguity can arise. Consider mapping the function $reverse : \forall A.\ LA \to LA$, that reverses the order of the elements of a given list, over a value of type $T(L(LA))$, for some type A. Both the mapping from the functor T and the mapping from the functor $T \circ L$ are well-typed. The fully explicit types of these instances of map are respectively:

$$\mathsf{map}_1[T][L(LA)][L(LA)]\ (reverse[LA]) : T(L(LA)) \to T(L(LA))$$

$$\mathsf{map}_1[T \circ L][LA][LA]\ (reverse[A]) : T(L(LA)) \to T(L(LA))$$

In the first case the lists of lists of A are reversed and in the second case the lists of A. Additional information on how a value is constructed is required to be able to pick the intended mapping. This ambiguity can be resolved by not allowing the types $(F \circ G)A$ and $F(GA)$ to be equal. In FML these types are only isomorphic:

$$(F \circ G)A \simeq F(GA)$$

We call this isomorphism a *canonical isomorphism*. The idea stems from shape theory, which will be discussed in more detail in Section 4.4. What is important to know here is that values of type FA can be uniquely represented by their shape, of type $F\mathbf{1}$, and their data, a list of values of type A. The type of the shape can therefore be identified by the functor F and the type of the data by the type A.

When we view the types in the isomorphism above in the light of shape theory, values of type $(F \circ G)A$ have a shape constructed from the functor $F \circ G$ and data of type A, and values of type $F(GA)$ have a shape constructed from the functor F and data of type GA. In FML this distinction manifests itself by the presence of two special combinators:

$$\begin{aligned} \mathsf{pack} &: F(GA) \to (F \circ G)A \\ \mathsf{unpack} &: (F \circ G)A \to F(GA) \end{aligned}$$

These combinators are the witnesses of the isomorphism above. The combinator pack is used as a constructor and unpack is used as a destructor with respect to a composite functor $F \circ G$. We illustrate this by giving a datatype definition in the the functional programming language Haskell:

```
data Comp f g a = Pack (f (g a))
```

Values of type `Comp f g a` and `f (g a)` are distinguished by the constructor function `Pack`. The destructor of the new datatype is the combinator unpack. It can also be defined in Gofer/Haskell-style:

```
unpack (Pack t) = t
```

FML provides pack and unpack combinators for the composition of functors of all arities, not just for functors of arity one. The combinators are subscripted with two numbers: the first number denotes the arity of the *outer* functor of the composition and the second number denotes the arity of the *inner* functor(s) of the composition. Consider for example the isomorphism with respect to the composition of a functor F of arity 2 with functor G and H of arity 1:

$$\begin{aligned} \mathsf{pack}_{2,1} &: F(GA, HA) \to F\langle G, H\rangle A \\ \mathsf{unpack}_{2,1} &: F\langle G, H\rangle A \to F(GA, HA) \end{aligned}$$

These combinators provide an isomorphism between values with a shape described by the functor F and data of types GA and HA, and values with a shape described by the functor $F\langle G, H\rangle$ and data of type A.

The types of pack and unpack for composite functors of arbitrary arity are thus:

$$\begin{array}{rcl} \mathsf{pack}_{m,n} & : & \forall F\!:\!m.\forall G_{i\in m}\!:\!n.\forall X_{j\in n}\!:\!T.\ F(G_i(\overline{X})_{i\in m}) \to F(\overline{G})(\overline{X}) \\ \mathsf{unpack}_{m,n} & : & \forall F\!:\!m.\forall G_{i\in m}\!:\!n.\forall X_{j\in n}\!:\!T.\ F(\overline{G})(\overline{X}) \to F(G_i(\overline{X})_{i\in m}) \end{array}$$

Again, these typings denote the types of a family of combinators.

The shape-data distinction also affects the use of the projection functors. One would expect that the types $\Pi^2_0\ (X, Y)$ and X are equal, and similarly that $\Pi^2_1\ (X, Y)$ and Y are equal. In FML, these types are only isomorphic. This means we have another canonical isomorphism regarding projection functors:

$$\Pi_i^{(m)}(X_1, .., X_m) \simeq X_i\ .$$

As for the first canonical isomorphism, FML provides two combinators to witness this isomorphism:

$$\begin{array}{rcl} \mathsf{tag}_{m,i} & : & X_i \to \Pi_i^m(X_1, .., X_m) \\ \mathsf{untag}_{m,i} & : & \Pi_i^m(X_1, .., X_m) \to X_i \end{array}$$

Similarly to the previous isomorphism, these combinators are used as constructors and destructors. Consider the projection functors Π^2_0 and Π^2_1, often called Exl and Exr. We define new datatypes in Haskell as follows:

```
data Exl a b = Tagl a
data Exr a b = Tagr b
```

The inverses can be defined thus:

```
untagl (Tagl t) = t
untagr (Tagr t) = t
```

Note that pack and tag are combinators that leave their arguments intact. They are merely used as labels to indicate that a value is constructed in a certain fashion. The combinators unpack and untag are only used to destruct these labels. It is however the presence of these combinators that allows the description of a mapping algorithm that doesn't require extra information by means of a functor argument.

Cons- and Snoc-lists. We shall illustrate the use of FML functors and combinators by describing cons-lists and snoc-lists in FML.

The functor that describes cons-lists is:

$$\mathit{ConsL} = \mu^1 + \langle \mathbf{1}\langle\rangle^2, \times\rangle^2$$

This functor is defined as the fix-point of the bifunctor $+\langle \mathbf{1}\langle\rangle^2, \times\rangle^2$, which is the composite of the functor $+$, the functor $\mathbf{1}\langle\rangle^2$ (a bifunctor that produces the unit type) and the functor $\times$.

We use some syntactic sugar in the definitions of the constructors for cons-lists. The notation $fx = t$ is syntactic sugar for $f = \lambda x.t$, and $f \circ g$ (function composition) is short for $\lambda x.f\ (g\ x)$. Function composition associates to the right. We will also denote the identity function $\lambda x.\ x$ with id. These notations are used throughout the rest of this chapter.

The usual constructors for cons-lists are defined as follows:

$$\begin{array}{rcl} \mathit{nil} & = & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0\ (\mathsf{pack}_{0,2}\ \mathsf{un}))) \\ \mathit{cons}\ x\ xs & = & (\mathsf{wrap}_1 \circ \mathsf{pack}_{2,2} \circ \mathsf{in}_1)\ (\mathsf{pair}\ x\ xs) \end{array}$$

Snoc-lists are defined as follows:

$$\mathit{SnocL} = \mu^1 + \langle \mathbf{1}\langle\rangle^2, \times\langle \Pi^2_1, \Pi^2_0\rangle^2\rangle^2$$

The functor *SnocL* is defined as the fix-point of the bifunctor $+\langle \mathbf{1}\langle\rangle^2, \times\langle \Pi^2_1, \Pi^2_0\rangle^2\rangle^2$. This bifunctor is the composite of the functor $+$, the functor $\mathbf{1}\langle\rangle^2$ and the functor $\times\langle \Pi^2_1, \Pi^2_0\rangle^2$, which is a bifunctor that swaps its arguments.

The constructors for snoc-lists are defined as follows:

$$\begin{array}{rcl} \mathit{nil} & = & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0\ (\mathsf{pack}_{0,2}\ \mathsf{un}))) \\ \mathit{snoc}\ xs\ x & = & (\mathsf{wrap}_1 \circ \mathsf{pack}_{2,2} \circ \mathsf{in}_1 \circ \mathsf{pack}_{2,2})\ (\mathsf{pair}\ (\mathsf{tag}_{2,1}\ xs)\ (\mathsf{tag}_{2,0}\ x)) \end{array}$$

Note that the empty snoc-list is identical to the empty cons-list. Also note that the arguments of *snoc* are *tagged*. The tags assure that the mapping algorithm, which is discussed below, can distinguish the head of a snoc-list (tagged with $\mathsf{tag}_{2,0}$) from its tail (tagged with $\mathsf{tag}_{2,1}$).

4.3 Mapping and Folding

The polytypic mapping algorithm in FML is defined by means of a set of reduction rules, which are added to the basic β-reduction and let-variant of β-reduction of the λ-calculus. The basic reduction rules are:

$$\begin{array}{rcl}(\lambda x.t_2)\ t_1 & > & t_2\{t_1/x\} \\ (\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 & > & t_2\{t_1/x\}\end{array}$$

Mapping. Functorial mapping for functors of arity m is implemented by means of the combinator map_m. The types of the map combinators are given by the following typing:

$$\mathsf{map}_m\ :\ \forall F\!:\!m.\forall X_{i\in m}, Y_{i\in m}\!:\!T.\ (X_i \to Y_i) \to_{i\in m} F(\overline{X}) \to F(\overline{Y})$$

The algorithm for polytypic mapping with respect to all supported functors of arity m, is defined by the reduction rules for the combinator map_m.

The algorithm for mapping with respect to composite functors is defined by reduction rules that consider terms constructed with the combinator pack. The reduction rule

$$\mathsf{map}_1\ f\ (\mathsf{pack}_{1,1}\ t)\ >\ \mathsf{pack}_{1,1}\ (\mathsf{map}_1\ (\mathsf{map}_1\ f)\ t)$$

for instance, implements the mapping of the composite functor $F \circ G$ of unary functors F and G. The $\mathsf{pack}_{1,1}$ combinator indicates to the mapping algorithm that the data is not of type GA, so the function f should not be applied to t at this point. The data has to be found 'deeper' inside the structure. This is done by applying $\mathsf{map}_1\ f$ to t.

The algorithm for mapping with respect to projection functors is defined by reduction rules that consider terms constructed with the combinator tag. The reduction rules

$$\begin{array}{rcl}\mathsf{map}_2\ f\ g\ (\mathsf{tag}_{2,0}\ t) & > & \mathsf{tag}_{2,0}\ (f\ t) \\ \mathsf{map}_2\ f\ g\ (\mathsf{tag}_{2,1}\ t) & > & \mathsf{tag}_{2,1}\ (g\ t)\end{array}$$

implement mappings with respect to functors Π^2_0 and Π^2_1. The tag combinators are used by the mapping algorithm as *tags* that indicate what function should be applied to a given datum. In this case, the combinator $\mathsf{tag}_{2,0}$ indicates that the function f, not g, should be applied to t, and the combinator $\mathsf{tag}_{2,1}$ indicates that the function g, not f, should be applied to t.

There are also reduction rules for mapping with respect to the functor constants $+, \times$ and $\mathbf{1}$. These reduction rules implement the mappings of the categorical sum, product and unit functors.

Here are all the reduction rules for map_m:

$$
\begin{array}{lcl}
\mathsf{map}_m\ f_{k\in m}\ (\mathsf{pack}_{n,m}\ t) & > & \mathsf{pack}_{n,m}\ (\mathsf{map}_n\ (\mathsf{map}_m\ \overline{f})\ t) \\
\mathsf{map}_m\ f_{k\in m}\ g\ (\mathsf{tag}_{m,i}\ t) & > & \mathsf{tag}_{m,i}\ (f_i\ t) \\
\mathsf{map}_2\ f\ g\ (\mathsf{pair}\ t\ u) & > & \mathsf{pair}\ (f\ t)\ (g\ u) \\
\mathsf{map}_2\ f\ g\ (\mathsf{in}_0\ t) & > & \mathsf{in}_0\ (f\ t) \\
\mathsf{map}_2\ f\ g\ (\mathsf{in}_1\ t) & > & \mathsf{in}_1\ (g\ t) \\
\mathsf{map}_0\ \mathsf{un} & > & \mathsf{un} \\
\mathsf{map}_m\ f_{k\in m}\ (\mathsf{wrap}_m\ t) & > & \mathsf{wrap}_m\ (\mathsf{map}_{m+1}\ \overline{f}\ (\mathsf{map}_m\ \overline{f})\ t)
\end{array}
$$

Note that the rule for type functors reduces a mapping of arity m to a mapping of arity $m{+}1$, which is the mapping of the functor that underlies the type functor. This rule is identical (but for notation) to the defining equation of `map` for type functors of arity one that we gave in Section 2.

Folding. The fold_m combinator is used to define the catamorphisms of an initial algebra with respect to the functor of arity $m+1$ that underlies an m-ary type functor. It is assigned the following type:

$$\mathsf{fold}_m\ :\ \forall F\!:\!m{+}1.\forall X_{i\in m}, Y\!:\!T.(F(\overline{X}, Y) \to Y) \to \mu^m F(\overline{X}) \to Y$$

The reduction of fold involves mapping, and is defined thus:

$$\mathsf{fold}_m\ f\ (\mathsf{wrap}_m\ t)\ >\ f\ (\mathsf{map}_{m+1}\ (\lambda x.x)_{i\in m}\ (\mathsf{fold}_m f)\ t)$$

The rule for fold_1 is identical (but for notation) to the defining equation for `cata2` that we gave in Section 2.

More Reduction Rules. The other reduction rules of FML concern the destruction of terms:

$$
\begin{array}{rcl}
\pi_0\ (\mathsf{pair}\ t_0\ t_1) & > & t_0 \\
\pi_1\ (\mathsf{pair}\ t_0\ t_1) & > & t_1 \\
\mathsf{case}\ f_0\ f_1\ (\mathsf{in}_j t) & > & f_j\ t \\
\mathsf{unpack}_{m,n}\ (\mathsf{pack}_{m,n}\ t) & > & t \\
\mathsf{untag}_{m,i}\ (\mathsf{tag}_{m,i}\ t) & > & t
\end{array}
$$

The destruction rule of recursive terms was already given by means of the rule for fold above.

Reductions of Snoc-lists. We show the reduction of the mapping of a function $f : A \to B$ over a snoc-list of type $Snoc\ A$. We use the definition of snoc-lists from Section 4.2.

Reduction of $\mathsf{map}_1\ f\ nil$ to nil proceeds as follows:

$$
\begin{array}{ll}
 & \mathsf{map}_1\ f\ nil \\
= & \\
 & \mathsf{map}_1\ f\ (\mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0\ (\mathsf{pack}_{0,2}\ \mathsf{un})))) \\
> & \\
 & \mathsf{wrap}_1\ (\mathsf{map}_2\ f\ (\mathsf{map}_1 f)\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0\ (\mathsf{pack}_{0,2}\ \mathsf{un})))) \\
> & \\
 & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{map}_2\ (\mathsf{map}_2\ f\ (\mathsf{map}_1 f))\ (\mathsf{map}_2\ f\ (\mathsf{map}_1 f))\ (\mathsf{in}_0\ (\mathsf{pack}_{0,2}\ \mathsf{un})))) \\
> & \\
 & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0 (\mathsf{map}_2\ f\ (\mathsf{map}_1 f)(\mathsf{pack}_{0,2}\ \mathsf{un})))) \\
> & \\
 & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0 (\mathsf{pack}_{0,2}\ (\mathsf{map}_0 \mathsf{un})))) \\
> & \\
 & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_0 (\mathsf{pack}_{0,2}\ \mathsf{un}))) \\
= & \\
 & nil
\end{array}
$$

The only reduction rules that are used in this reduction are the rules for mapping.

The reduction of a non-empty snoc-list $\mathsf{map}_1\ f\ (snoc\ xs\ x)$ to $snoc\ (\mathsf{map}_1\ f\ xs)\ (f\ x)$ proceeds as follows:

$$
\begin{array}{ll}
 & \mathsf{map}_1\ f\ (snoc\ xs\ x) \\
> & \\
 & \mathsf{map}_1\ f\ (\mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{pair}\ (\mathsf{tag}_{2,1}\ xs)\ (\mathsf{tag}_{2,0}\ x)))))) . \\
> & \\
 & \mathsf{wrap}_1\ (\mathsf{pack}_{2,2}\ (\mathsf{in}_1 (\mathsf{pack}_{2,2}\ (\mathsf{pair}\ (\mathsf{tag}_{2,1}\ (map_1\ f\ xs))\ (\mathsf{tag}_{2,0}\ f\ x))))) , \\
> & \\
 & snoc\ (\mathsf{map}_1\ f\ xs)\ (f\ x)
\end{array}
$$

This reduction uses the rules for mapping as well as β-reduction.

The two reductions above show that the usual defining equations for mapping over snoc-lists, i.e:

```
map f Nil          = Nil
map f (Snoc xs x) = Snoc (map f xs) x
```

can be derived with the FML reduction rules.

Reduction with Haskell In Section 4.2 we demonstrated the meaning of FML combinators by providing Haskell `data` declarations and function definitions. In Appendix B we show the definition of a subset of FML terms in Haskell. The subset includes all combinators with respect to functors of arities 0, 1 and 2. The Haskell reduction system can be used to reduce these terms. Polytypic combinators are defined using qualified kinds.

4.4 Semantics

The intended semantics for FML interprets functors as *functors shapely over lists*, which is a notion that stems from shape theory (see [Jay95]). Informally, a functor F is shapely over lists if values of type FA can be uniquely identified by their shape of type $F1$ and their content of type $List(A)$. The idea of canonical isomorphisms and their associated combinators originates from interpreting type constructors as functors shapely over lists. The functors that are currently supported by FML are all shapely over lists. Other functors than that are shapely over lists are for example arrays and matrices. These functors can be added to the FML type system as functor constants and interact with the other functors without type-theoretic problems. If the proper reduction rules for mapping with respect to these new functors are added as well, it is possible to define the mapping algorithm for arrays of trees, trees of matrices or other combinations.

4.5 Traversals

FML functors are intended to be interpreted as functors shapely over lists, so it makes sense to introduce combinators for the explicit separation of shape and data and for the combination of shape and data. In PolyP we were able to define these combinators in terms of traversals (see Section 3.4). Traversal are parameterized by an arbitrary monad, which is possible in PolyP because of the qualified kinds of the Haskell type system. Qualified kinds are not supported by FML, so we define traversals with respect to a specific monad: the *parser monad*. This monad is a combination of a state transformer monad and an exception monad. It mimics a global state and the possibility of aborting the computation by means of an exception. See Appendix A for a definition of state transformer monads in Haskell.

In FML, we define parser monads as follows:

$$\begin{aligned} P_S(X) &= S \to (X \times S) + \mathbf{1} \\ \mathit{return} &= \lambda x.\lambda s.\ \mathsf{in}_0\ (\mathsf{pair}\ x\ s) \end{aligned}$$

$$
\begin{aligned}
bind \;=\; & \lambda m.\lambda f.\lambda s.\ \mathsf{case}\ (m\ s) \\
& \qquad (\lambda p.\ f\ (\pi_0\ p)\ (\pi_1\ p)) \\
& \qquad (\lambda x.\ \mathsf{in}_1\ \mathsf{un})
\end{aligned}
$$

Left traversals with respect to this monad are assigned the following typing in FML:

$$
\begin{aligned}
ltraverse_m : \ & \forall S{:}T.\forall F{:}m.\forall X_{i\in m}{:}T.\forall Y_{i\in m}{:}T. \\
& \overline{(X \to P_S(Y))} \to F(\overline{X}) \to P_S(F(\overline{Y}))
\end{aligned}
$$

The reduction rules for this combinator are:

$$
\begin{array}{rcl}
ltraverse_0\ \mathsf{un} & > & return\ \mathsf{un} \\
ltraverse_2\ f_0\ f_1\ (\mathsf{pair}\ t_0\ t_1) & > & bind\ (f_0\ t_0)\ (\lambda y_0. \\
& & bind\ (f_1\ t_1)\ (\lambda y_1. \\
& & return\ \mathsf{pair}\ y_0\ y_1)) \\
ltraverse_2\ f_0\ f_1\ (\mathsf{in}_j\ t) & > & bind\ (f_j\ t)\ (\lambda y. \\
& & return\ (\mathsf{in}_j\ y)) \\
ltraverse_m\ f_{i\in m}\ (\mathsf{tag}_{m,i}\ t) & > & bind\ (f_i\ t)\ (\lambda y. \\
& & return\ (\mathsf{tag}_{m,i}\ y)) \\
ltraverse_n\ f_{i\in n}\ (\mathsf{pack}_{m,n}\ t) & > & bind\ (ltraverse_m\ (ltraverse_n\ \overline{f})_{i\in m}\ t)\ (\lambda y. \\
& & return\ (\mathsf{pack}_{m,n}\ y)) \\
ltraverse_m\ f_{i\in m}\ (\mathsf{wrap}_m\ t) & > & bind\ (ltraverse_{m+1}\ \overline{f}\ (ltraverse_m\ \overline{f}\ t)\ (\lambda y. \\
& & return\ (\mathsf{wrap}_m\ y))
\end{array}
$$

Right traversal can be defined by the combinator $rtraverse_m$ with the same type as $ltraverse_m$. The reduction rules are also the same, except for the rule for pairs which is:

$$
\begin{array}{rcl}
rtraverse_2\ f_0\ f_1\ (\mathsf{pair}\ t_0\ t_1) & > & bind\ (f_1\ t_1)\ (\lambda y_1. \\
& & bind\ (f_0\ t_0)\ (\lambda y_0. \\
& & return\ (\mathsf{pair}\ y_0\ y_1)))
\end{array}
$$

The order in which a right traversal visits the data of a structure is the opposite of the order that a left traversal uses.

Shape and Data Decomposition. Using left traversal we can define a function that separates the shape and data of a given structure:

$$
separate_m : \forall F{:}m.\forall X_{i\in m}{:}T.\ F(\overline{X}) \to F(\overline{\mathbf{1}}) \times (LX_0 \times (LX_1 \times (\ldots \times LX_{m-1}))
$$

We can implement $separate_m$ as follows:

$$
\begin{aligned}
separate_m\ t \;=\; & \mathsf{case}\ (ltraverse_m\ \overline{f}\ t\ nil) \\
& id \\
& (\lambda x.\ \mathsf{pair}\ (\mathsf{map}_m\ (\lambda y.\mathsf{un})\ t)\ nil)
\end{aligned}
$$

The third argument of the case combinator in this definition is only used when the left traversal *fails*. This never happens, so the argument is actually obsolete. Its sole purpose is to complete the case statement.

The functions $\overline{f}$ are defined thus:

$$
\begin{aligned}
& f_i : X_i \to P_{LX_0 \times \ldots \times LX_{m-1}} \mathbf{1} \\
& f_i\ x\ s \;=\; \mathsf{in}_0\ (\mathsf{pair}\ \mathsf{un}\ (\mathsf{map}_m\ \overline{g}\ s))
\end{aligned}
$$

The inverse of $separate_m$ can be defined as a right traversal. We can define a combinator

$$
combine_m : \forall F\!:\!m. \forall X_{i \in m}\!:\!T.\ F(\mathbf{1}_{i \in m}) \times (LX_0 \times \ldots \times LX_{m-1}) \to F(\overline{X}) + \mathbf{1}
$$

that takes a shape and a list of data and fills the shape with the data using a right traversal. If there is not enough data to fill the shape, the computation of the traversal fails.

$$
combine_m\ p \;=\; rtraverse_m\ \overline{h}\ (\pi_0\ p)\ (\pi_1\ p)
$$

The functions $\overline{h}$ are defined thus:

$$
\begin{aligned}
& h_i : \mathbf{1} \to P_{LX_0 \times \ldots \times LX_{m-1}} X_i \\
& h_i\ t\ s \;=\; \mathsf{case}\ (\mathsf{unpack}_{2,2}\ (\mathsf{unwrap}_1\ (\pi_{m,i} s))) \\
& \qquad\qquad (\lambda x.\ \mathsf{in}_1\ \mathsf{un}) \\
& \qquad\qquad (\lambda xs.\ \mathsf{in}_0\ (\mathsf{pair}\ (\pi_0 xs)\ (\mathsf{map}_m\ (k_j\ xs)_{j \in m}\ s)))
\end{aligned}
$$

where $k_j : LX_i \to LX_j \to LX_j$ is given by $k_i\ xs\ ys = xs$ when $i = j$ and $k_j\ xs\ ys = ys$ otherwise.

The combinators $\pi_{m,i}$ denote projections from an m-ary product to the i-th element of that product, which can be defined using the binary product functor $\times$ and its projection functions π_0 and π_1.

$$
\pi_{m,i} : \forall X_{j \in m}\!:\!T. X_0 \times (X_1 \times (\ldots X_{m-1})) \to X_i
$$

The combinator unwrap_m is the inverse of wrap_m and is defined thus:

$$
\mathsf{unwrap}_m \;=\; \mathsf{fold}_m(\mathsf{map}_{m+1}\ \overline{\mathsf{id}}\ \mathsf{wrap}_m)
$$

4.6 Polytypic Definitions

In PolyP, the user can define his or her own polytypic functions by induction on the structure of functors. The same feature is desirable FML too. It provides the user with the means to define the combinators for mapping, folding, traversing and other combinators (i.e. for zipping) using recursive definitions.

This means that FML should be extended with a fix-point combinator to handle recursive definitions. It is however not possible to define the combinator map_m with just a fix-point combinator, because map_m is not just defined in terms of itself, but also in terms of mapping for functors with different arities. Furthermore, we need induction on the structure of functors. These problems are solved by extending the type system with quantification over arities and with a term-constructor poly (similar to the `polytypic` construct of PolyP) that performs case analysis on functors. This extension is called FML_{poly} and is discussed in more detailed in [JBM98].

For illustrational purposes we give the definition of *ltraverse* by means of the poly construct. It is defined using an auxiliary combinator *plt*, which defines left traversal with rearranged arguments. This is necessary because the first argument of a function that is defined with poly should be a term that is constructed with F. The type of the auxiliary combinator is:

$$\begin{aligned}&\mathit{plt} : \forall m : N.\forall S{:}T.\forall F{:}m.\forall X_{i\in m}{:}T.\forall Y_{i\in m}{:}T.\\ &\qquad F(\overline{X}) \to (X \to P_S(Y))_{i\in m} \to P_S(F(\overline{Y}))\end{aligned}$$

Notice that this type is quantified over arities.

Left traversal with the poly construct is defined thus:

$$\begin{array}{llcll}
\multicolumn{5}{l}{\mu\ \mathit{ltrav}.\mathsf{let\ poly}}\\
& \mathit{plt}_0\ \mathsf{un} & = & \mathit{return}\ \mathsf{un} & |\\
& \mathit{plt}_2\ (\mathsf{pair}\ x_0\ x_1) & = & \lambda f_0, f_1.\ \mathit{bind}\ (f_0\ t_0)\ (\lambda y_0. & \\
& & & \qquad\qquad \mathit{bind}\ (f_1\ t_1)\ (\lambda y_1. & \\
& & & \qquad\qquad \mathit{return}\ \mathsf{pair}\ y_0\ y_1)) & |\\
& \mathit{plt}_2\ (\mathsf{in}_0 x) & = & \lambda f_0, f_1.\ \mathit{bind}\ (f_0\ t)\ (\lambda y. & \\
& & & \qquad\qquad \mathit{return}\ (\mathsf{in}_0\ y)) & |\\
& \mathit{plt}_2\ (\mathsf{in}_1 x) & = & \lambda f_0, f_1.\ \mathit{bind}\ (f_1\ t)\ (\lambda y. & \\
& & & \qquad\qquad \mathit{return}\ (\mathsf{in}_1\ y)) & |\\
& \mathit{plt}_m\ (\mathsf{tag}_{m,i}\ x) & = & \lambda f_{i\in m}.\ \ \mathit{bind}\ (f_i\ t)\ (\lambda y. & \\
& & & \qquad\qquad \mathit{return}\ (\mathsf{tag}_{m,i}\ y)) & |\\
& \mathit{plt}_m\ (\mathsf{pack}_{n,m}\ x) & = & \lambda f_{i\in m}.\ \ \mathit{bind}\ (\mathit{ltrav}_m\ (\mathit{ltrav}_n\ \overline{f})_{i\in m}\ t)\ (\lambda y. & \\
& & & \qquad\qquad \mathit{return}\ (\mathsf{pack}_{n,m}\ y)) & |
\end{array}$$

$$
\begin{array}{lcl}
plt_m\ (\mathsf{wrap}_m\ x) & = & \lambda f_{i \in m}.\ \ bind\ (ltrav_{m+1}\ \overline{f}\ (ltrav_m\ \overline{f}\ t)\ (\lambda y. \\
 & & \quad return\ (\mathsf{wrap}_m\ y)) \qquad |
\end{array}
$$
$$
\mathsf{in}\ (\Lambda m.\lambda f_{i \in m}.\lambda x.plt_m\ x\ \overline{f})
$$

Notice that the body of the **let poly** construction uses explicit abstraction over arities, which is part of FML_{poly}.

The poly construct can be used to define a shape compression algorithm similar to the PolyP algorithm discussed in Section 3.5. Shape decompression is problematic though, because it requires explicit knowledge of the functor of the compressed shape, or a single type to describe all possible shapes.

4.7 Conclusions

FML is an extension of the Hindley-Milner type system with functors. Functors are added as type constructors and types can be quantified by functors of a certain arity.

A set of combinators that includes constructors and destructors for all the functors is added to the basic terms of HM. The constructors and destructors for composite and projection functors are witnesses to the *canonical* isomorphisms, which allow terms to express their decomposition into shape and data. This decomposition makes the description of uniform mapping and folding algorithms possible.

We have seen that the basic set of combinators can be extended with combinators for traversals, which can be used to define functions that separate the shape and data of terms explicitly. It is also possible to extend FML with user-defined polytypic definitions. Combinators for mapping and traversing can be expressed as polytypic definitions, but shape decompression for example, is problematic.

The FML functors are intended to be interpreted as functors shapely over lists. Currently FML supports regular functors of any arity, but the set of base functors can be extended to include other functors that are shapely over lists like arrays. Another desirable extension of FML is the inference of subscripts of combinators.

FML has not yet been implemented, but in Appendix B we show how to implement a finite subset of FML terms in Haskell.

Chapter 5

Nested Datatypes

In the previous sections we have focussed primarily on regular datatypes for a couple of reasons. Firstly they are the most common in programming practice. Secondly, they can be described as initial algebras in a categorical framework. This gives us the polytypic notion of a catamorphism and provides a semantics that supports calculational reasoning. Thirdly, regular datatypes are defined inductively, so we can also define polytypic functions inductively.

In this Section we shall part with regularity and discuss so-called *nested datatypes*. These datatypes have been neglected in functional programming until recently. Now people are discovering some of their theoretical importance and practical applications (see also [BM98]). To be precise, we shall model nested datatypes by higher-order functors in the Haskell dialect Hugs 1.3c [Jon98] and discuss the pros and cons of this approach.

5.1 What is a Nested Datatype?

Recall from Section 3.1 that a datatype `D` is regular if all uses of `D` on the right hand side of its definition are applied to the same parameters as on the left hand side. For the definition of nested datatypes this requirement is dropped.

Consider the following declaration of a nested datatype in Haskell:

```
data Nest a = NilN | ConsN a (Nest (a,a))
```

This datatype is not regular due to the application of the type parameter of `Nest` to `(a,a)`. Values of types that are constructed with `Nest` (*nests*) have a similar structure as cons-lists, but the entries are "squared" each step down the structure.

Examples of values of type `Nest Int` are:

```
NilN
ConsN 6 NilN
ConsN 6 (ConsN (3,4) (ConsN ((5,7),(1,8))))
```

or less verbosely, using commas and brackets:

```
[]
[6]
[6,(3,4),((5,7),(1,8))]
```

The third nest has three entries, containing a total of seven integers.

5.2 Nested Datatypes as Higher-Order Functors

In Sections 2 and 3.1 we have seen that in Haskell, a regular datatype can be described as the fix-point of a functor. For example, the datatype

```
data List a = Nil | Cons a (List a)
```

can be described as follows:

```
data FList a b = Nil | Cons a b

type List a = Mu (FList a)
```

The type constructor `FList` can be extended to a functor by providing a suitable `map` function and we can provide a function `cata` to define recursive functions on lists. These functions and the definition of `Mu` are omitted here but can be found in Section 2.

Note that the functor `FList` is obtained from the original `data` declaration of `List` by replacing all references to `List a` by a new type parameter `b`. The parameter of a nested datatype however, is not always instantiated with the same type. This means we cannot refer to each instantiation with a single type parameter `b`.

When for example the definition of the datatype `Nest` is unrolled once, `Nest` is instantiated with `(a,a)`. When we unroll `Nest` a second time, it is instantiated with `((a,a),(a,a))`. These two parameter types are different and cannot be replaced by a single type parameter `b`.

Although the representation of nested datatypes as fixed points of functors fails, we can still describe nested datatypes in a categorical way by using *higher-order functors*.

5.2.1 Higher-Order Functors

A higher-order functor is a functor in the category that has functors as objects. The arrows of this category are given by so-called *natural transformations*. In functional programming, a natural transformation `eta` from functor `f` to functor `g` is a polymorphic function of the following type:

```
eta :: (Functor f, Functor g) => f a -> g a
```

In category theory, the definition of a natural transformation is more general and can be any collection of arrows, not just collections defined by single polymorphic definitions. In general, natural transformations should satisfy a so-called naturality requirement. For `eta` this requirement is:

```
map f . eta = eta . map f
```

In words this requirement states: the action of `eta` on a given value of type `f a`, does not affect individual elements (of type `a`) of that value. Whenever `eta` is polymorphic, this requirement always holds.

Just like any other functor, a higher-order functor consists of two mappings: a mapping from objects to objects, i.e. from functors to functors, and a mapping from arrows to arrows, i.e. from natural transformations to natural transformations.

The object mapping of a higher-order functor can be modeled by higher-order type constructors in Haskell, i.e. type constructors of kind `(* -> *) -> * -> *`. In order to define the arrow mapping of a higher-order functor, it must be possible to use a polymorphic function (natural transformation) as the argument of a function, and still be able to use that argument in a polymorphic way. This kind of polymorphism, which is called *rank 2* polymorphism, is not allowed in standard Haskell because of the so-called *monomorphic type restriction* that is inherited from the Hindley-Milner type system. In the Hugs 1.3c implementation however, functions are allowed to have rank 2 polymorphic type signatures. These type signatures have an explicit universal quantifier (`forall`) in the argument type. This feature of Hugs 1.3.c is exploited to define the arrow mappings of higher-order functors.

A higher-order functor in Haskell is defined to be a higher-order type constructor that is a member of the following class:

```
class HOFunctor hf where
  homap :: (Functor f, Functor g) =>
           (forall a. f a -> g a) -> hf f b -> hf g b
  mapf  :: Functor f => (a -> b) -> hf f a -> hf f b
```

The function `homap` implements the arrow mapping of a higher-order functor `hf`. We assume that `homap` satisfies the functor laws (see 2). Note that `homap` accepts a rank 2 type argument that represents a natural transformation. It also returns a natural transformation (a polymorphic function), since `b` is implicitely universally quantified.

Whenever the object mapping of a higher-order functor `hf` is applied to a functor `f`, the result is a new functor `hf f`. The function `mapf` implements the arrow mapping of this functor. This is made explicit by the following `instance` definition:

```
instance (HOFunctor hf, Functor f) => Functor (hf f) where
  map = mapf
```

Another way to define this mapping is to provide separate instances for each higher-order functor that is defined. We prefer our approach because it requires less explicit context information.

We will now present a higher-order semantics for nested datatypes by describing them as fix-points of higher-order functors. First we show how to describe the regular datatype of cons-lists with a higher-order functor `HOFList`:

```
data Functor f => HOFList f a = Nil | Cons a (f a)
```

The type constructor `HOFList` accepts a functor argument `f` and a type argument `a` and returns values of type `HOFList f a`. We extend `HOFList` to a higher-order functor by defining instances of `homap` and `mapf`:

```
instance HOFunctor HOFList where
   homap eta = \x -> case x of
                       Nil      -> Nil
                       Cons a l -> Cons a (eta l)
   mapf f    = \x -> case x of
                       Nil      -> Nil
                       Cons a l -> Cons (f a) (map f l)
```

In order to define cons-lists in terms of `HOFList`, a higher-order fix-point constructor `HOMu` is required to replace the first-order `Mu` that is used to define fix-points of standard functors. We define `HOMu` thus:

```
data HOMu hf a = HOIn (hf (HOMu hf) a)
```

This definition makes `HOMu hf` denote a fix-point of `hf`.

The datatype of cons-lists can now be defined as a fix-point of the higher-order functor `HOFList`:

```
type List = HOMu HOFList
```

The constructor `HOMu` constructs a functor from a higher-order functor. The arrow mapping of the constructed functor is defined thus:

```
instance HOFunctor hf => Functor (HOMu hf) where
  map f (HOIn x) = HOIn (map f x)
```

The function `map` on the right hand side of this definition corresponds to the arrow mapping of the functor `hf (HOMu hf)`. This mapping is implemented by the function `mapf` as we have seen before.

The strength of this higher-order approach to defining datatypes lies in the fact that it does not only apply to regular datatypes such as `List`, but also to nested datatypes. We can define the nested datatype `Nest`, as defined above, alternatively as the fix-point of a higher-order functor thus:

```
data Functor f => HOFNest f a = NilN | ConsN a (f (a,a))

instance HOFunctor HOFNest where
   homap eta = \x -> case x of
                       NilN      -> NilN
                       ConsN a n -> ConsN a (eta n)
   mapf f    = \x -> case x of
                       NilN      -> NilN
                       ConsN a n -> ConsN (f a) (map (f -*- f) n)
type Nest = HOmu HOFNest
```

5.2.2 Catamorphisms

Datatypes that are constructed from a higher-order functor with the higher-order fix-point constructor `HOMu` can be described as initial algebras with respect to base category `Fun`, i.e. as higher-order initial algebras. This means that we can define (higher-order) catamorphisms for these datatypes.

We define a combinator `hocata` for the construction of higher-order catamorphisms as follows:

```
hocata :: (HOFunctor hf, Functor f) =>
          (forall a. hf f a -> f a) -> HOmu hf b -> f b
hocata phi = phi . homap (hocata phi) . hoOut
```

The natural transformation `hoOut` is the inverse of `HOIn`. It is defined thus:

```
hoOut :: HOMu hf a -> hf (HOMu hf) a
hoOut (HOIn x) = x
```

Note that the definition of `hocata` is very similar to that of `cata` in section 2. The difference is that `hocata` works only for natural transformations, i.e. polymorphic functions, whilst `cata` works with ordinary functions.

As an example of a higher-order catamorphism, consider the function `flattenNest` that flattens nests to ordinary lists. We define `flattenNest` as a higher-order catamorphisms in terms of the higher-order algebra `fflattenNest` as follows:

```
flattenNest :: Nest a -> [a]
flattenNest = hocata fflattenNest

fflattenNest :: HOFNest [] a -> [a]
fflattenNest NilN        = []
fflattenNest (ConsN x n) = x : concat (map tau n)
  where tau (x,y) = [x,y]
```

Note that `flattenNest` uses the built-in Haskell datatype of lists. The Haskell function `concat` implements the flattening of a list of lists to a list.

We mentioned above that the difference between higher-order catamorphisms and standard catamorphisms is that the former work exclusively for natural transformations. As a consequence, the expressive power of higher-order catamorphisms is limited. They cannot be used to express simple functions like `sum` (for summing a structure of integers), and `concat` (for concatenating a list of lists), that can be expressed as standard catamorphisms. In the next Section we define a class of nested datatypes and a polytypic crush operator for all datatypes in that class. With this crush operator we can partly solve the problem concerning expressive power.

5.3 A Class of Nested Datatypes

We introduce a class of unary nested datatypes that can be described inductively. The class consists of nested datatypes that are constructed as fix-points of higher-order functors. The grammar of higher-order functors we use, is derived from the grammar of PolyP functors (see Section 3.1):

```
HOFunctor ::= HOSum     hf hg
            |  HOProduct hf hg
            |  HOEmpty
            |  HOPar
            |  HORec     hf
            |  HOComp    d  hf
            |  HOConst   t
```

The non-terminals hf and hg describe higher-order functors, d describes datatype constructors, which should also be functors, and t describes ordinary types.

The noticable difference with the PolyP functors is that the higher-order functor constructor HORec, which replaces PolyP's bifunctor Rec, has a higher-order functor parameter hf. This parameter is used to encode the actual nesting of a nested datatype.

The higher-order functors above are defined pointwise by the following Haskell data declarations:

```
data Functor f =>
   HOEmpty          f a = Un

data Functor f =>
   HOPar            f a = Par  a

data (Functor f, HOFunctor hf) =>
   HORec     hf     f a = Rec  (f (hf f a))

data (Functor f, HOFunctor hf, HOFunctor hg) =>
   HOSum     hf hg f a = Sum  (Either (hf f a) (hg f a))

data (Functor f, HOFunctor hf, HOFunctor hg) =>
   HOProduct hf hg f a = Pair (hf f a, hg f a)

data (Functor f, Functor d, HOFunctor hf) =>
   HOComp     d hf f a = Comp (d (hf f a))

data Functor f =>
   HOConst          f t = Con t
```

In all declarations, the functor parameter f is passed on to the composite higher-order functors. Only in the declaration of HORec, it is actually applied. It is not directly applied to the parameter a, as would have been the case if we wanted to define only regular datatypes. The functor `hf f` is applied to `a` before `f`, making nesting possible.

All unary regular datatypes can be expressed with these higher-order functors. Consider the following definitions of cons-lists and rose trees for instance:

```
type List = HOMu (HOSum HOEmpty
                        (HOProduct HOPar
                                   (HORec HOPar)))
type Rose = HOMu (HOProduct HOPar
                            (HOComp List
                                    (HORec HOPar)))
```

In both these declarations, the higher-order functor `HORec HOPar` is used to describe the recursive part of the definition. This reflects the fact that in the body of the definition of a regular datatype, the datatype constructor (referred to by `HORec`) may only be instantiated with the type parameter (referred to by `HOPar`) that was used on the left hand side of the definition. Whenever we define a datatype using `HORec` with an argument that is different from `HOPar`, we obtain a nested datatype. Consider for instance the following definition of the datatype of nests:

```
type Nest = HOMu (HOSum HOEmpty
                        (HOProduct HOPar
                                   (HORec (HOProduct HOPar
                                                     HOPar))))
```

In this case `HORec` is instantiated with the higher-order functor `HOProduct HOPar HOPar`. This reflects the fact that the recursive constituents of nests have tupled arguments.

Given the definition of the datatype `List` above, we can define the usual constructors for lists as follows:

```
nil :: List a
nil = (In . Sum . Left) Un

cons :: a -> List a -> List a
cons x n = (In . Sum . Right . Pair) (Par x, Rec n)
```

A similar definition of the constructors `nilN` and `consN` for the datatype `Nest` is more complicated. The definition of `nilN` is no problem, it is identical to that of `nil`:

```
nilN :: Nest a
nilN = (In . Sum . Left) Un
```

For the definition of `consN` we need to transform values of type `Nest (a,a)` to values of type `HORec (HOProduct HOPar HOPar) Nest a`. These types are isomorphic, but not equal. Note that the values of the former type have shapes of type `Nest` and data of type `(a,a)`, where as values of the latter type have shapes of type `HORec (HOProduct HOPar HOPar) Nest` and data of type `a`. The transformation is performed by mapping the function `Pair . (Par -*- Par)` over the value of type `Nest (a,a)` and applying the constructor `Rec` to the result. Thus `consN` is a linear function.

```
consN :: a -> Nest (a,a) -> Nest a
consN x n = (In . Sum . Right . Pair)
                  (Par x, Rec (map (Pair . (Par -*- Par)) n))
```

The `data` declarations above describe the actions of the higher-order functors on the level of objects (i.e. functors). The following `instance` declarations make the higher-order functors member of the class `HOFunctor` and provide their actions on the level of arrows (i.e. natural transformations) by means of the function `homap`. The function `mapf`, that describes the arrow mapping of the functor that is obtained by applying a higher-order functor to a given functor, is defined simultaneously.

```
instance HOFunctor HOEmpty where
  homap eta Un = Un
  mapf  f   Un = Un

instance HOFunctor HOPar where
  homap eta (Par x) = Par x
  mapf  f   (Par x) = Par (f x)

instance HOFunctor hf => HOFunctor (HORec hf) where
  homap eta (Rec x) = Rec (eta (map (homap eta) x))
  mapf  f   (Rec x) = Rec (map (mapf f) x)

instance (HOFunctor hf, HOFunctor hg) =>
         HOFunctor (HOSum hf hg) where
  homap eta (Sum x) = Sum ((homap eta -+- homap eta) x)
  mapf  f   (Sum x) = Sum ((mapf f    -+- mapf  f)   x)

instance (HOFunctor hf, HOFunctor hg) =>
```

```
         HOFunctor (HOProduct hf hg) where
  homap eta (Pair (x,y)) = Pair (homap eta x, homap eta y)
  mapf  f   (Pair (x,y)) = Pair (mapf  f   x, mapf  f   y)

instance (Functor d, HOFunctor hf) =>
         HOFunctor (HOComp d hf) where
  homap eta (Comp x) = Comp (map (homap eta) x)
  mapf  f   (Comp x) = Comp (map (mapf f) x)

instance HOFunctor (HOConst t) where
  homap eta (Con t) = Con t
  mapf  f   (Con t) = Con t
```

5.3.1 A Polytypic `flatten`

The function `flatten`, which was previously defined for the datatype `Nest`, can be defined for all the nested datatypes in our class. As before, `flatten` is defined as a higher-order catamorphism of a function `fflatten`. We define a Haskell class `Flatten` to be a subclass of the class `HOFunctor` and provide a function `fflatten` for each of the higher-order functors. The function `flatten` is defined in the declaration of the class `Flatten`.

```
class HoFunctor hf => Flatten hf where
  fflatten :: hf [] a -> [a]
  flatten :: HOMu hf a -> [a]

  flatten = hocata fflatten
```

Here are the definitions of `fflatten` for all higher-order functors:

```
instance Flatten HOEmpty where
  fflatten Un = []

instance Flatten HOPar where
  fflatten (Par x) = [x]

instance Flatten hf => Flatten (HORec hf) where
  fflatten (Rec x) = concat (map fflatten x)

instance (Flatten hf, Flatten hg) =>
         Flatten (HOSum hf hg) where
```

```
  fflatten (Sum x) = (fflatten `either` fflatten) x

instance (Flatten hf, Flatten hg) =>
         Flatten (HOProduct hf hg) where
  fflatten (Pair (x,y)) = fflatten x ++ fflatten y

instance (Flatten hf, Flatten hg) =>
         Flatten (HOComp (HOMu hf) hg) where
  fflatten (Comp x) = (concat . flatten . map fflatten) x

instance Flatten (HOConst t) where
  fflatten (Con t) = []
```

The definition of `flatten` uses the built-in Haskell datatype of lists and not the higher-order definition of lists that we gave before. The reason for this is that we use the functions `(++)`, for appending two lists, and `concat`, for concatenating a list of lists to a single list. In the standard semantics, these functions are defined as catamorphisms. It is however not possible to define them as higher-order catamorphisms in the higher-order semantics, see [BM98] for details.

5.3.2 A Polytypic `crush` Combinator

We know that for regular datatypes we can define a combinator `crush` that, given a binary operator and a seed value, 'crushes' the data of its argument to a single value. Basic functions for flattening a structure, summing the contents of a structure or testing the membership of a given value to a structure, can be defined with `crush`. Another term for functions defined with `crush` is *reductions*. See [Mee96] for a detailed discussion on the crush combinator for regular datatypes.

One way of transforming the concept of reductions to nested datatypes in the higher-order semantics is to abstract from the datatype of lists in the definition of `flatten` above. We replace the datatype `[]` with an arbitrary monad `m` that has a plus and a zero. In the Haskell prelude we have the following predefined classes for such monads:

```
class Monad m => MonadZero m where
    zero   :: m a

class MonadZero m => MonadPlus m where
    (++)   :: m a -> m a -> m a
```

The operator `(++)` must be associative, and `zero` must be its neutral element.

Now we can define a polytypic function `mcrush` by declaring a constructor class `MCrush`:

```
class HOFunctor hf => MCrush hf where
    mfcrush :: (MonadPlus m, Functor m) => hf m a -> m a
    mcrush  :: (MonadPlus m, Functor m) => HOMu hf a -> m a

mcrush = hocata mfcrush
```

We instantiate `MCrush` for an arbitrary nested datatype by defining the function `mfcrush` for each nested datatype constructor. The definitions, which are omitted here, are obtained from the definitions of `fflatten` by replacing the empty list by `zero`, `(++)` for lists by `(++)` for monads, a singleton list by the application of `return` and `concat` by `join`.

Note that in Haskell, the class `Monad` is not a subclass of `Functor`, although theoretically monads are always functors. This means that we have to explicitly state that `m` is functor in the contexts of the type signatures of `mfcrush` and `mcrush`.

This abstraction gives us a polytypic function on nested datatypes for each monad with a plus and zero. By instantiating the list monad, we obtain `flatten`. Another possibility is to use the monad `Maybe`, as defined in the Haskell prelude:

```
data Maybe a = Nothing | Just a

return x      = Just x
join Nothing  = Nothing
join (Just m) = m

zero         = Nothing
Nothing ++ y = y
x       ++ y = x
```

The function `mcrush` that is associated with this monad returns the first element of a structure or `Nothing` if the structure is empty.

At this point we know of no other useful monads that lead to useful polytypic functions. We do know that we cannot define `sum` with `mcrush`, because of the naturality requirement.

A simple way of defining reductions like `sum` for nested datatypes, is by assuming the existence of list catamorphisms. Given a polytypic `flatten` and list catamorphisms, we can define a `crush` combinator for nested datatypes by first flattening and then crushing the resulting list with a list catamorphism:

```
crush :: HOFunctor hf => (a -> a -> a) -> a -> HOMu hf a -> a
crush op e = foldr op e . flatten
```

We assume `flatten` returns a Haskell list. The combinator `foldr` implements catamorphisms on Haskell lists.

Now we can define a polytypic `sum` for nested datatypes as follows:

```
sum :: HOFunctor hf => HOMu hf a -> [a]
sum = crush (+) 0
```

5.4 Higher-Order PolyP?

PolyP extends Haskell with a `polytypic` construct which allows the user to define polytypic functions by induction on the structure of regular bifunctors. Together with the facility to derive the underlying bifunctor of a unary regular datatype, this makes it possible to define polytypic functions for regular datatypes in PolyP.

It would be nice to extend PolyP in a similar fashion with respect to nested datatypes defined by the higher-order functors we presented above. This extension of Haskell, let's call it HO-PolyP, should have a `hopolytypic` construct to define polytypic functions by induction on the structure of higher-order functors and should be able to generate higher-order functors from unary nested datatypes.

In HO-PolyP, the definition of `flatten` would be much shorter and easier to understand than its definition using constructor classes:

```
flatten :: Nested d => d a -> [a]
flatten = hocata fflatten

hopolytypic fflatten :: HOFunctor hf => hf [] a -> [a]
  = case hf of
      hg + hh -> fflatten `either` fflatten
      hg * hh -> (++) . fflatten -*- fflatten
      Empty   -> \x -> []
      Par     -> \x -> [x]
      Rec hg  -> concat . pmap fflatten
      d @ hg  -> concat . flatten . pmap fflatten
      Const t -> \x -> []
```

We have renamed the higher-order functor names to the same names as PolyP uses for bifunctors. They still refer to higher-order functors though.

The definition above is almost identical to the one for `flatten` in PolyP. The most obvious differences are the extra parameter in the `Rec` case, which allows the application of `flatten` to nested datatypes, and the type signatures which refer to nested datatypes and higher-order functors.

It should be possible to define most polytypic functions that can be defined for regular datatypes in PolyP, for nested datatypes in HO-PolyP. We could for instance define monadic functions like traversals. It should also be possible to define the algorithm for shape compression that we implemented in PolyP in Section 3.5.

We have not investigated the details concerning a possible implementation of HO-PolyP, but at a glance it seems to be possible. The implementations of type-checking and the generation of higher-order functors in HO-PolyP may prove to be more complicated than PolyP's implementations however.

5.5 Conclusions

Most approaches to polytypy focus on regular datatypes. In this chapter we have shown how to expand the polytypic horizon to include a larger class of datatypes, including regular and nested datatypes.

We have shown how to encode regular and nested datatypes with higher-order functors and how to define polytypic functions for these datatypes in Hugs1.3c, a dialect of Haskell that incorporates rank 2 polymorphism.

The higher-order catamorphisms that come with this higher-order semantics, prove to be less expressive than catamorphisms in the standard semantics. Despite this lack of expression we have defined polytypic reductions for nested datatypes by assuming the existence of standard list catamorphisms.

The higher-order semantics that we have presented here is also discussed in an article by Bird and Meertens ([BM98]), but they do not give an inductive class of nested datatypes and so they cannot define a polytypic crush function as we did. They do propose a more general type of reduction that can be applied to the nested datatype `Nest`:

The following nest for example,

```
[6,(3,4),((5,7),(1,8))]
```

can be reduced with two operators, `(+)` and `(*)`, to:

```
6 + (3 * 4) + ((5 * 7) * (1 * 8))
```

It remains to be seen if generalized reductions of this kind can be defined generically for our class of nested datatypes.

Chapter 6

Conclusions

The notion of polytypy arises when we abstract from datatypes, so the first question one should ask is when discussing polytypy is: what is a datatype? Throughout this thesis we have adopted the idea that datatypes are described by categorical functors, but each chapter considered a different class of functors.

In PolyP we considered only regular bifunctors that were used to describe unary regular datatypes as initial algebras. PolyP has been implemented as an extension of Haskell, so the concept of functors describing datatypes is placed on top of the Haskell type system. PolyP generates functors for datatypes from the Haskell datatype declarations. Theoretically, functors come before datatypes, but in PolyP datatypes come before functors. Despite of this lack of theoretical foundation, PolyP is a practical tool for defining polytypic functions in Haskell. We demonstrated this by implementing an algorithm for compression and decompression of unary regular datatypes. Especially the possibility of combining polytypic and monadic programming in PolyP is quite powerful.

Functorial ML places functors at the core of its type system. FML functors are intended to be interpreted as functors shapely over lists, which include regular functors, but also functors for constructing arrays and matrices. Currently FML only supports regular functors (of arbitrary arity though), but other functors that are shapely over lists can be added as functor constants in the future. Compared to PolyP programs, FML programs look rather awkward because of the extensive use of combinators, but these combinators make the definition of polytypic functions in FML possible. FML has not yet been implemented, but we have briefly shown how to translate FML to Haskell in Appendix B.

PolyP and FML can currently not describe nested datatypes, because they describe recursive datatypes as fix-points of standard functors. Describing recursive datatypes as fix-points of higher-order functors includes the possibility of describing nested datatypes, but the catamorphisms of non-nested datatypes are replaced by the less powerful higher-order catamorphisms. The higher-order approach seems to be well-suited for polytypic purposes and it might be possible to implement a higher-order version of PolyP that models datatypes as fix-points of higher-order functors and supports a `hopolytypic` construct. All datatypes in the class of nested datatypes that we introduced, are shapely over lists, so it might be possible to add nested datatypes to FML as well.

Appendix A

State Transformer Monads in Haskell

```
data State s a = ST (s -> (a,s))

instance Functor (State s) where
    map f (ST st) = ST (\s -> let (x,s') = st s in (f x, s'))

instance Monad (State s) where
    return x    = ST (\s -> (x,s))
    ST m >>= f  = ST (\s -> let (x,s') = m s
                                ST f'  = f x
                            in  f' s')

updateState :: (s -> s) -> State s s
updateState f = ST (\s -> (s, f s))

fetchState :: State s s
fetchState = updateState id

startingWith :: State s a -> s -> a
ST m `startingWith` s0 = result where (result,_) = m s0
```

Appendix B

Implementing FML in Haskell

B.1 The Functors

```
-- The Functor Classes
class Functor0 f where
  map0 :: f -> f

class Functor1 f where
  map1 :: (a -> b) -> f a -> f b

class Functor2 f where
  map2 :: (a -> b) -> (c -> d) -> f a c -> f b d

-- Projection Functors
data Pi10 a             = Tag10 a
data Pi20 a b           = Tag20 a
data Pi21 a b           = Tag21 b

-- Composite Functors
data Functor0 f => Comp00 f           = Pack00 f
data Functor0 f => Comp01 f       a   = Pack01 f
data Functor0 f => Comp02 f       a b = Pack02 f
data (Functor1 f, Functor0 g) =>
                   Comp10 f g         = Pack10 (f g)
data (Functor1 f, Functor1 g) =>
                   Comp11 f g     a   = Pack11 (f (g a))
data (Functor1 f, Functor2 g) =>
                   Comp12 f g     a b = Pack12 (f (g a b))
data (Functor2 f, Functor0 g1, Functor0 g2) =>
```

```
                   Comp20 f g1 g2     = Pack20 (f g1 g2)
data (Functor2 f, Functor1 g1, Functor1 g2) =>
                   Comp21 f g1 g2 a   = Pack21 (f (g1 a) (g2 a))
data (Functor2 f, Functor2 g1, Functor2 g2) =>
                   Comp22 f g1 g2 a b = Pack22 (f (g1 a b) (g2 a b))

-- Type Functors
data Functor1 f => Mu0 f              = Wrap0 (f (Mu0 f))
data Functor2 f => Mu1 f a            = Wrap1 (f a (Mu1 f a))

-- Functor Constants
data Sum a b            = In0 a | In1 b
data Product a b        = Pair a b
data One                = Un
```

B.2 Mapping

```
-- Projection Functors

instance Functor1 Pi10 where
  map1 f (Tag10 t) = Tag10 (f t)

instance Functor2 Pi20 where
  map2 f g (Tag20 t) = Tag20 (f t)

instance Functor2 Pi21 where
  map2 f g (Tag21 t) = Tag21 (g t)

-- Composite Functors

instance Functor0 f => Functor2 (Comp02 f) where
  map2 f g (Pack02 t) = Pack02 (map0 t)

instance (Functor1 f, Functor1 g) => Functor1 (Comp11 f g) where
  map1 f (Pack11 t) = Pack11 (map1 (map1 f) t)

instance (Functor1 f, Functor2 g) => Functor2 (Comp12 f g) where
  map2 f g (Pack12 t) = Pack12 (map1 (map2 f g) t)

instance (Functor2 f, Functor1 g1, Functor1 g2)
      => Functor1 (Comp21 f g1 g2) where
```

```
  map1 f (Pack21 t) = Pack21 (map2 (map1 f) (map1 f) t)

instance (Functor2 f, Functor2 g1, Functor2 g2)
      => Functor2 (Comp22 f g1 g2) where
  map2 f g (Pack22 t) = Pack22 (map2 (map2 f g) (map2 f g) t)

-- Type Functors

instance Functor1 f => Functor0 (Mu0 f) where
  map0 (Wrap0 t) = Wrap0 (map1 map0 t)

instance Functor2 f => Functor1 (Mu1 f) where
  map1 f (Wrap1 t) = Wrap1 (map2 f (map1 f) t)

-- Functor Constants

instance Functor2 Sum where
  map2 f g (In0 t) = In0 (f t)
  map2 f g (In1 t) = In1 (g t)

instance Functor2 Product where
  map2 f g (Pair t u) = Pair (f t) (g u)

instance Functor0 One where
  map0 Un = Un
```

B.3 Folding

```
fold0 f (Wrap0 t) = f (map1 (fold0 f) t)
fold1 f (Wrap1 t) = f (map2 (\x -> x) (fold1 f) t)
```

B.4 Other Reduction Rules

```
-- Destruction w.r.t. Composite and Projection Functors
-- (Canonical isomorphisms)

unpack00 (Pack00 t) = t
unpack01 (Pack01 t) = t
unpack02 (Pack02 t) = t
unpack10 (Pack10 t) = t
```

```
unpack11 (Pack11 t) = t
unpack12 (Pack12 t) = t
unpack20 (Pack20 t) = t
unpack21 (Pack21 t) = t
unpack22 (Pack22 t) = t

untag10 (Tag10 t) = t
untag20 (Tag20 t) = t
untag21 (Tag21 t) = t

-- Destruction w.r.t. Sum and Product Functors

pi0 (Pair x y) = x
pi1 (Pair x y) = y

--junc implements FML's case combinator
junc f g (In0 t) = f t
junc f g (In1 t) = g t
```

B.5 Example: Cons-lists and Snoc-lists

```
type List     = Mu1 (Comp22 Sum (Comp02 One) Product)
type SnocList = Mu1 (Comp22 Sum (Comp02 One)
                                (Comp22 Product Pi21 Pi20))

--nil :: List a
--nil :: SnocLis a
nil = (Wrap1 . Pack22 . In0 . Pack02) Un

cons :: a -> List a -> List a
cons x xs = (Wrap1 . Pack22 . In1) (Pair x xs)

snoc :: SnocList a -> a -> SnocList a
snoc xs x
  = (Wrap1 . Pack22 . In1 . Pack22) (Pair (Tag21 xs) (Tag20 x))

-- Summing

sumCons :: List Int -> Int
sumCons
  = fold1 (junc (\x -> 0) plus . unpack22)
```

```
sumSnoc :: SnocList Int -> Int
sumSnoc
  = fold1 ( junc (\x -> 0)
                 ( plus
                 . map2 untag21 untag20
                 . unpack22
                 )
          . unpack22
          )

plus :: Product Int Int -> Int
plus (Pair x y) = x + y

-- Reduction Examples

sumCons (cons 3 (cons 4 nil)) -> 7
sumSnoc (snoc (snoc nil 5) 6) -> 11
```

Bibliography

[BJM96] G. Bellè, C.B. Jay, and E. Moggi. Functorial ML. In K. Herbert and S.D. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs*, Lecture Notes in Computer Science, 1140. Springer-Verlag, 1996.

[BM98] Richard S. Bird and Lambert Meertens. Nested datatypes. In *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, June 1992.

[Hag87] T. Hagino. *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh, 1987.

[Jan97] Patrik Jansson. Functional polytypic programming — use and implementation. Technical report, Chalmers University of Technology, Gothenburg, Sweden, 1997. Licentiate thesis. Available from `http://www.cs.chalmers.se/~patrikj/lic/`.

[Jay95] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995. Also in ESOP '94.

[JBM98] C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. Extended version of [BJM96] in press for JFP'98, 1998.

[JJ97] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[Jon93] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA '93*, pages 52–61, 1993.

[Jon94] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[Jon98] Mark P. Jones. A technical summary of the new features in hugs 1.3c. Available from `ftp://ftp.cs.nott.ac.uk/ haskell/hugs/hugs13/WhatsNew`, March 1998.

[Mee86] L. Meertens. Algorithmics — towards programming as a mathematical activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, volume 1 of *CWI Monographs*, pages 289–334. North–Holland, 1986.

[Mee96] L. Meertens. Calculate polytypically! In *PLILP'96*, volume 1140 of *LNCS*, pages 1–16. Springer Verlag, 1996.

[MH95] E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Conference Record of FPCA '95, SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture*, pages 324–333, 1995.