

Dealing with Large Bananas

Ralf Lämmel¹, Joost Visser¹, and Jan Kort²

¹ CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

² University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Abstract. Many problems call for a mixture of generic and specific programming techniques. We propose a polytypic programming approach based on generalised (monadic) folds where a separation is made between basic fold algebras that model generic behaviour and updates on these algebras that model specific behaviour. We identify particular basic algebras as well as some algebra combinators, and we show how these facilitate structured programming with updatable fold algebras. This blend of genericity and specificity allows programming with folds to scale up to applications involving large systems of mutually recursive datatypes. Finally, we address the possibility of providing generic definitions for the functions, algebras, and combinators that we propose.

1 Introduction

Polytypic programming [JJ97,Hin00] aims at relieving the programmer from repeatedly writing functions of similar functionality for different user-defined datatypes. For example, for any datatype parametric in α , ‘crushing’ the values of type α in a given structure can be defined *fully* generically [Mee96,Hin99]. Such a generic function abstracts from constructors. It is defined by induction on the structure of datatypes in terms of sums, products and others.

Many problems rather call for a *mixture* of generic and specific programming techniques. Think of a program transformation. On the one hand, it must implement specific behaviour for particular constructs of the language at hand. On the other hand, it acts on the remaining constructs in a completely generic way: it preserves them. Or think of a program analysis. It often follows a completely generic scheme such as accumulation or reduction, while usually only a few patterns require specific functionality. This interplay of genericity and specificity has also been observed by others (e.g., [Vis00]).

To address this mixture of genericity and specificity, we propose a polytypic programming approach based on generalised [Fok92,MFP91] and monadic [Fok94,MJ95] folds for systems of mutually recursive datatypes. It is generally accepted that programming with folds (or, more generally, with morphisms) is desirable because it imposes ‘structured programming’, it facilitates (optimizing) program transformation, it untangles traversal schemes from traversal-specific ingredients, and it facilitates reasoning about programs. Programming with folds offers a restricted form of generic programming, in the sense that traversal schemes such as fold functions can be defined generically for large classes of datatypes. Recent research has focused on extending the class of permitted datatypes, and on identifying the various traversal schemes and their properties [Mee92,FSS92,SF93,MH95,BP99].

Yet, programming with generalised folds is not truly generic because actual programming means to pass algebras to the fold function. These algebras provide the ingredients of the actual traversal, and their structure depends on the actual datatype. Thus, while the traversal schemes might be generic, their instantiations are obtained through non-generic programming.

We propose to separate constructing fold algebras into (i) obtaining a generic fold algebra, through polytypic programming and/or reuse from a library of basic fold algebras and algebra combinators, and (ii) updating the generic algebra with specific behaviour for particular constructors. This separates the places where one wants to be generic from the places where one needs to be specific. Since both algebras and updates on them are regarded as first-class citizens, structured programming with them is facilitated. In particular, we identify some generic functions for calculating with monadic folds.

Our approach can be used, for example, for the development of program transformations and analyses in the context of legacy system renovation [CC90,BSV00], where one is concerned with the adaptation of large software applications, for example written in COBOL. The sheer size of the underlying languages in this area makes some sort of generic programming indispensable; defining traversals on the language’s syntax non-generically is simply not feasible. Yet, for particular constructors specific behaviour must be specified. Programming with folds scales up to these kinds of problems when a functional language is used that provides, as we propose, generalised folds for mutually recursive datatypes and a combinator language for fold algebras, including a mechanism for updating generic algebras with specific behaviour.

Section 2 briefly recapitulates the various elements involved in existing methods of programming with folds. Section 3 explains the separation of generic algebras and algebra updates, which is the key to scalable programming with folds. Section 4 extrapolates this separation to monadic folds. Throughout these sections, a running Haskell example, adapted from [MJ95], is used to identify and illustrate the required elements for programming with updatable folds. Section 5 provides a more abstract formulation of our approach, including polytypic definitions of some elements.

2 Programming with folds

Using an example adapted from [MJ95], we will quickly recapitulate the various elements involved in existing methods of programming with folds. Moreover, we will explain the lack of scalability of these methods.

Remarks We use Haskell examples throughout the paper. In particular, we use Haskell 98 extended with multi-parameter type classes, which are supported by the main Haskell implementations. We use classes to overload functions merely for convenience — our treatment does not rely on them. We chose not to use ‘functional dependencies’ [Jon99] in class headers (as in: `class Fold alg t a | alg t -> a where ...`). This would make more accurate overloading resolution possible, allowing the user to write fewer explicit types, but it is currently only supported by Hugs. In Section 4 on monadic folds, we make use of stackable monads from Andy Gill’s Monad Template Library, to be found via <http://www.haskell.org>.

2.1 An example

When using folds, a programmer writes functions consuming values of a datatype D in terms of a fold function which captures the recursive traversal scheme for D . The fold function is parameterized by a *fold algebra*, which holds as many functions as there are constructors in the datatype. These functions are meant to replace the constructors in the traversal.

Example 1. Assume for example the following system of datatypes, which represents the abstract syntax of a simple functional language:

```
data Type = TVar String
          | Arrow Type Type
data Expr = Var String
          | Apply Expr Expr
          | Lambda (String,Type) Expr
```

The type of the algebras that parameterize folds over this system of datatypes is the following:

```
data Cata a b = Cata{ tvar    :: String -> a
                    , arrow  :: a -> a -> a
                    , var    :: String -> b
                    , apply  :: b -> b -> b
                    , lambda :: (String,a) -> b -> b }
```

The algebra type is named `Cata` because the corresponding fold functions capture the *catamorphic* scheme of recursion. We will comment on paramorphisms [Mee92] in Section 5. We use a flat Haskell record to model an algebra for usability reasons. There are other possible encodings. Some of them will be discussed in Section 6.

The family of fold functions for the system of datatypes can be represented by the following class and instance declarations:

```
class Fold alg t a where
  fold :: alg -> t -> a
instance Fold (Cata a b) Type a where
  fold alg (TVar x)    = (tvar alg) x
  fold alg (Arrow s t) = (arrow alg) (fold alg s) (fold alg t)
instance Fold (Cata a b) Expr b where
  fold alg (Var x)      = (var alg) x
  fold alg (Apply f a)  = (apply alg) (fold alg f) (fold alg a)
  fold alg (Lambda (x,t) b) = (lambda alg) (x,(fold alg t)) (fold alg b)
```

Note that in general the fold functions can be mutually recursive just like the system of datatypes. Given these definitions, a programmer can begin to write functions consuming values of one of the datatypes, by passing appropriate algebra values to one of the fold functions. For instance, a function for constant function elimination can be written as follows:

```
cfe    :: Expr -> Expr
cfe    = fold cfeAlg
cfeAlg :: Cata Type Expr
cfeAlg = Cata{ tvar    = TVar
              , arrow  = Arrow
              , var    = Var
              , apply  = \f a -> case f of
                          (Lambda (x,t) b) -> if not (elem x (freevars b))
                                                then b
                                                else (Apply f a)
                          _ -> (Apply f a)
              , lambda = Lambda }
```

The function `freevars` can be programmed in the same style, as we will show in Section 3.3.

2.2 Scalability problems

Imagine using the technique of programming with folds, not for the toy language of Example 1, which has a syntax definition with two nonterminals (types) and five productions (constructors), but for COBOL, which has a syntax definition with several hundreds of nonterminals and productions. This occurs in the application areas of program analysis and transformation such as legacy system renovation [CC90,BSV00]. There are several problems with respect to scalability:

Initial effort Before programming with folds can begin, the algebra type and the fold functions need to be defined. Since both the number of field declarations in the algebra type, and the number of function equations are equal to the number of constructors, the effort involved is proportional to the size of the syntax definition.

Repeated effort Instantiating a fold function with an algebra almost requires as much effort as writing a traversal from scratch. The number of field definitions in the fold algebra is again equal to the number of constructors. So, no matter how small the problem to be solved by a traversal, the size of the algebra to be written is proportional to the size of the syntax definition.

In principle, the first problem can be solved by generating folds (refer, e.g., to [BB85,She91]), offering them as language primitives (as, e.g., in Charity [CS92]), or providing polytypic definitions for them (as, e.g., in PolyP [JJ97]). However, there are some problems with the existing approaches regarding systems of mutually recursive datatypes and the kind of algebra notion supported by them. In Section 5, we attempt to improve on these existing approaches. To solve the second problem, this paper proposes to separate generic fold algebras from language-specific updates on them. This is explained in Sections 3 and 4.

3 Programming with updatable fold algebras

We propose to separate the construction of fold algebras into (i) obtaining a basic algebra, and (ii) updating the algebra. This separates the places where one wants to be generic from the places where one needs to be specific. In this section, we will explain how programming with updatable fold algebras proceeds, and we will identify some useful basic fold algebras. In Section 4, some sophistication is added to the technique of programming with folds by accommodating monads and (monadic) fold algebra combinators. Finally, in Section 5, the generic structures involved in programming with updatable fold algebras are given generic definitions.

3.1 Updating algebras

To explain the separation into basic algebras and updates, we revisit Example 1.

Example 2. The algebra `cfeAlg` can be constructed by applying a fold algebra update to a basic fold algebra. In this particular case, the basic fold algebra `idmap` is appropriate:

```
idmap :: Cata Type Expr
idmap = Cata{ tvar    = TVar
             , arrow  = Arrow
             , var    = Var
             , apply  = Apply
             , lambda = Lambda }
```

The generic behaviour captured by this algebra is to traverse a term without changing it, i.e., fold applied to `idmap` is the identity function. This holds because constructors are replaced by themselves. This is a law each fold should satisfy [MJ95]. In order to obtain `cfeAlg` from `idmap`, we apply the update `cfeUpd`:

```
cfeAlg = cfeUpd idmap
cfeUpd alg = alg{ apply = \f a -> case f of
                          (Lambda (x,t) b) -> if not (elem x (freevars b))
                                                then b
                                                else (Apply f a)
                          _ -> Apply f a }
```

Here we make use of the Haskell syntax `r{a1=x1, . . . ,an=xn}` for record update.

The separation of a basic fold algebra and an update on it, is the key to making programming with folds scalable. The basic fold algebra, which is proportional to the size of the language’s syntax definition, can be derived automatically or defined polytypically (see Section 5). The update needs to contain problem-specific functionality only, and is provided by the programmer.

3.2 Type-preserving and type-unifying

The basic fold algebra `idmap` and all algebras obtained by updating it are *type-preserving* in the sense that when folding with them a `Type` is mapped to a `Type`, and an `Expr` is mapped to an `Expr`. This is captured by the following type synonym:

```
type Preserve = Cata Type Expr
```

Type-preserving algebras are useful for programming (program) transformations. Another important class of algebras are the *type-unifying* ones. These map both `Expr` and `Type` onto the same result type. This is captured as follows:

```
type Unify a = Cata a a
```

The next subsection features such type-unifying algebras. As will become clear, type-unifying algebras are useful for programming (program) analyses.

3.3 Crushing

We start our discussion of type-unifying basic fold algebras with the parameterized basic fold algebra `crush`.

```
crush :: a -> (a->a->a) -> Unify a
crush e o = Cata{ tvar      = \x    -> e
                  , arrow   = \a b -> a 'o' b
                  , var     = \x    -> e
                  , apply   = \a b -> a 'o' b
                  , lambda  = \(x,t) b -> t 'o' b }
```

The parameters of this algebra, i.e., the value `e` and the binary operator `o`, are assumed to form a monoid. Alternatively, a type class `Monoid` could have been used here. Instantiation of `crush` would then proceed by type specialisation instead of passing parameters explicitly. The name `crush` is inspired by the related concept of polytypic crushing on parameterized datatypes [Mee96,Hin99]. Polytypic crushing means to collect and to reduce all values of type α in a datatype parametric in α . In contrast, our `crush` has to be updated before it collects values in a given data structure at all. The basic algebra just defines the reduction of intermediate results. Given a term `t`, the expression `fold (crush e o) t` will be evaluated to `e`, if we assume the monoid unit laws. This type of reduction does not depend on the parameterization of a datatype.

Example 3. To demonstrate the use of the type-unifying parameterized algebra `crush`, we will define a program analysis that collects free variables. First we instantiate `crush` to obtain a basic fold algebra `collect`:

```
collect = crush [] (++)
```

Then we define a collector of variables:

```
vars      :: Expr -> [String]
vars      = fold (varsUpd collect)
varsUpd alg = alg{ var = \x -> [x] }
```

And we derive a collector of free variables as needed in Example 1:

```
freevars  :: Expr -> [String]
freevars  = fold (fvUpd collect)
fvUpd alg = (varsUpd alg){ lambda = \(x,t) b -> filter (x/=) b }
```

This two-step update illustrates the modularisation of algebra updates. In Section 4 another technique is discussed. Of course, `collect` could have been updated in two points (i.e., constructors) at once.

Example 4. Another use of `crush` is to build a basic fold algebra `count` for counting:

```
count = crush 0 (+)
```

A counter of variables is constructed by updating `count`:

```
countvars :: Expr -> Integer
countvars = fold (cvUpd count)
cvUpd alg = alg{ var = \x -> 1 }
```

4 Merging monads and updatable folds

There are several reasons for using monads in combination with updatable fold algebras. Firstly, monadic effects can be used to address issues such as context propagation (environment monad), side-effects (I/O and state monad), and failure (error monad). Secondly, monadic updatable folds can be used to elegantly modularise programs.

4.1 Monadic folds

Monadic folds are explained in [Fok94,MJ95]. Some variants are discussed in [MBJ99]. The monadic algebra type and type synonyms for type-preserving and type-unifying monadic algebras for our example language are as follows:

```
data Monad m => MCata m a b
  = MCata{ mtvar   :: String -> m a
          , marrow  :: a -> a -> m a
          , mvar    :: String -> m b
          , mapply  :: b -> b -> m b
          , mlambda :: (String,a) -> b -> m b }
```

```
type MPreserve m = MCata m Type Expr
type MUnify m a  = MCata m a a
```

For brevity we give the monadic fold function only for `Type`; for `Expr` it is similar:

```
instance Monad m => Fold (MCata m a b) Type (m a) where
  fold alg (TVar x)    = (mtvar alg) x
  fold alg (Arrow s t) = do {s' <- fold alg s; t' <- fold alg t; marrow alg s' t'}
```

Note that the traversal scheme modelled by the monadic fold function explicitly sequences the computations of the recursive calls.

4.2 Lifting fold algebras

Monadic algebras can be constructed via two routes. Either directly, by updating a monadic basic fold algebra, or indirectly, by updating an ordinary algebra and *lifting* it to a monadic one. The lifting operator `unit` is straightforwardly defined as follows:

```
unit      :: Monad m => Cata a b -> MCata m a b
unit alg = MCata{ mtvar    = \x    -> return ((tvar alg) x)
                  , marrow  = \a b  -> return ((arrow alg) a b)
                  , mvar    = \x    -> return ((var alg) x)
                  , mapply  = \f a  -> return ((apply alg) f a)
                  , mlambda = \xt b -> return ((lambda alg) xt b) }
```

Of course, if the programmer wishes to use monadic effects in particular updates, only the direct route is available. As we will show, indirectly constructed updates and directly constructed ones can be composed, so the programmer is not forced to deal with monads where he does not use them.

4.3 Fold algebra composition

The algebra update `cfeUpd` of Example 2 is not quite suitable to be merged (by function composition) with other updates, because the fall-through arm of the case and the `else` branch of the conditional explicitly rebuild the original term. It would override the functionality specified by previous updates for all application nodes, not just for constant function applications. To prepare

this update for modular composition, it could instead refer to the algebra `alg` that is being updated (substitute `apply alg` for `Apply`). There is another technique which facilitates merging of updates. It is based on an algebra combinator `plus` and a neutral algebra `zero`.

```
plus :: MonadPlus m => MCata m a b -> MCata m a b -> MCata m a b
plus s s'
  = MCata{ mtvar    = \x      -> ((mtvar s) x)      'mplus' ((mtvar s') x)
        , marrow   = \a b    -> ((marrow s) a b)   'mplus' ((marrow s') a b)
        , mvar     = \x      -> ((mvar s) x)       'mplus' ((mvar s') x)
        , mapply   = \f a    -> ((mapply s) f a)   'mplus' ((mapply s') f a)
        , mlambda  = \ (x,t) b -> ((mlambda s) (x,t) b) 'mplus' ((mlambda s') (x,t) b) }
```

```
zero :: MonadPlus m => MCata m a b
zero = MCata{ mtvar    = \x      -> mzero
        , marrow   = \a b    -> mzero
        , mvar     = \x      -> mzero
        , mapply   = \f a    -> mzero
        , mlambda  = \xt e -> mzero }
```

These employ a monad with `plus` and `zero` (backtracking or error monad) to model the success or failure of algebra members. For convenience we additionally define an algebra combinator `try`, which tries to apply a type-preserving algebra and resorts to `idmap` when it fails:

```
try  :: MonadPlus m => MPreserve m -> MPreserve m
try s = s 'plus' (unit idmap)
```

Note that in this definition `idmap` is lifted to obtain a monadic `idmap`.

Example 5. The function `cfe` can now be reformulated.

```
cfe  :: Expr -> Maybe Expr
cfe  = fold (try cfeAlg :: MPreserve Maybe)

cfeAlg :: MonadPlus m => MPreserve m
cfeAlg = zero{ mapply = \f a -> case f of
    (Lambda (x,t) b)
      -> do guard (not (elem x (freevars b)))
            return b
    _ -> mzero }
```

Algebras formulated as updates on `zero` can freely be combined with other (appropriately typed) algebras by means of the combinators `plus` and `try`. This will be illustrated below.

4.4 Carried monads

It is well known that monadic folds are not expressive enough for all effects in traversals [MJ95]. The reason for this is that the sequencing of recursive calls which is weaved into the monadic fold function sometimes needs to be modified. In these cases, monads can be used in a different way, which we call *carried* (vs. *weaved-in*). We introduce the following type synonyms for carried monadic fold algebras:

```
type PreserveM m = Cata (m Type) (m Expr)
type UnifyM m a = Unify (m a)
```

Note that in carried monadic fold algebras, the sequencing of recursive calls needs to be done explicitly by the programmer. We can define `unit`, `zero`, `plus` and `try` for carried monadic algebras too. As in the weaved-in case, carried monadic algebras can be constructed directly or by lifting ordinary algebras. We will postfix names with `M` to indicate that carried monads are involved.

4.5 Casting weaved-in to carried monadic fold algebras

For some effects, carried monads are necessary, but in general they are more cumbersome than weaved-in monads, because the programmer is burdened with sequencing. Also, the restricted expressiveness of weaved-in monads yields more theorems for free. Fortunately, we can define a function `carried` that casts a weaved-in monadic algebra to a carried one.

```
carried :: Monad m => MCata m t e -> Cata (m t) (m e)
carried alg
  = Cata{ tvar      = \x          -> mtvar alg x
        , arrow    = \ma mb     -> do {a <- ma; b <- mb; marrow alg a b}
        , var      = \x          -> mvar alg x
        , apply    = \mf ma     -> do {f <- mf; a <- ma; mapply alg f a}
        , lambda   = \(x,mt) mb -> do {t <- mt; b <- mb; mlambda alg (x,t) b} }
```

The following example shows how `carried` can be used to resort to carried monads only for effects that need them.

Example 6. We define an algebra for performing substitutions. An environment (or reader) monad is used to propagate a context of type `Subst = [(String,Expr)]`. A state monad is used to generate new variable names, which are needed to prevent variable capture.

```
lookupAlg :: (MonadPlus m, MonadReader Subst m) => MPreserve m
lookupAlg = zero{ mvar = \x -> mlookup x }
restoreAlg :: (MonadPlus m, MonadReader Subst m, MonadState Int m) => PreserveM m
restoreAlg = zeroM{ lambda = \(x,mt) mb -> do env <- ask
                                             x' <- new_name
                                             t <- mt
                                             b <- restore ((x,Var x'):env) mb
                                             return (Lambda (x',t) b) }
substAlg :: (MonadPlus m, MonadReader Subst m, MonadState Int m) => PreserveM m
substAlg = tryM (carried lookupAlg 'plusM' restoreAlg')
```

The algebra `lookupAlg` takes care of the actual substitution of a variable. It is defined as a weaved-in monadic algebra. The algebra `restoreAlg` takes care of adding a renaming of a bound variable to the context *before* processing the body of a lambda abstraction. Here, a *carried* monad is needed. In the algebra `substAlg`, these two algebras are combined into a carried algebra, by first casting the weaved-in monadic algebra to a carried one, and then applying `plusM`.

5 Generic bananas

In the foregoing sections, we gave Haskell definitions of the ingredients for programming with (monadic) updatable folds: the fold algebra type, the fold functions, the basic fold algebras `idmap`, `crush` and `zero`, the fold combinators `unit` and `plus`, and the casting function `carried`. These definitions were *specific* to our example system of datatypes.

Of course, to truly enable *generic* programming, programmers should not be burdened with repeatedly supplying such definitions for all systems of datatypes that come up. In this section, we will demonstrate that generic definitions of the ingredients of programming with updatable folds can be given. These definitions can be implemented by a program generator, or by supplying them as language primitives in a functional language. This would allow generic programming *with* updatable folds. Alternatively, a generic programming language which allows these definitions to be expressed, would additionally enable programming *of* updatable folds.

5.1 Systems of datatypes

In the polytypic definitions to come, we use a flavour of polytypic programming [JJ97,Hin00]. We will perform induction over the structure of systems of (mutually recursive) datatypes. This structure is given by the following grammar:

$$\begin{array}{ll}
S ::= \emptyset \mid N = D \mid S \cup S & \text{-- systems of datatypes} \\
D ::= C T \mid D + D & \text{-- datatype definitions} \\
T ::= 1 \mid T \times T \mid N & \text{-- type expressions} \\
N & \text{-- names of datatypes} \\
C & \text{-- constructor names}
\end{array}$$

We use s , d , t , n , and c , possibly subscripted or primed, to range over respectively S , D , T , N , and C . For convenience, we introduce the notation $c(s)$ to denote the type of the constructor c in the system s , i.e., if $n = \dots + c t + \dots \in s$, then $c(s) = t \rightarrow n$.

As the grammar details, a system of datatypes s is a set of equations, a datatype definition d is a sum of types, labeled with constructor names, and a type expression t is a product over names of datatypes. Three features of this grammar are noteworthy. Firstly, constructor names are not suppressed in the representation of datatype definitions. Indeed, constructor names are indispensable when generic programming is to be mixed with specific programming. Secondly, the grammar explicitly distinguishes datatypes from type expressions. If they would be merged into a single nonterminal, that allows both sums and products, unintended expressions would be generated, e.g., sums not qualified with constructors, or constructors occurring inside products. Finally, though constructors are usually typed in a curried fashion, we use products for the parameters of constructors. This allows a more homogeneous treatment as common in polytypic programming. We only consider complete and non-extensible systems of datatypes in this paper. For the moment being, we limit ourselves to non-parameterized datatypes without function types and nested sums involved. At the end of the section we will discuss whether these limitations can be lifted.

5.2 Fold algebras

We need to define the fold algebra type induced by a system s of datatypes. This is a generalisation of the algebra type for a single datatype, which is well understood. Since we want to abstract from the concrete structure of algebras (whether they are records or tuples, flat or nested), we will provide a (semi-formal) axiomatisation of fold algebras. The Haskell approach of the previous sections should be regarded as one model of this axiomatisation.

Intuitively, the algebra type of a datatype system s is obtained as a collection of function types derived from all the constructor types in s by consistently replacing names of datatypes by distinct type variables. To accommodate type variables, we define type schemes $TS \supset T$, i.e., type expressions which may contain type variables. Type schemes are defined according to the following grammar:

$$\begin{array}{ll}
TS ::= 1 \mid TS \times TS \mid N \mid X & \text{-- type schemes} \\
X & \text{-- type variables}
\end{array}$$

We use τ and α to range respectively over TS and X . Now we can proceed to define s -fold algebras. \mathcal{A} is an s -fold algebra for a system s of datatype definitions if:

1. For each equation $n = d$ in s there is a type scheme $\bar{n}(\mathcal{A})$ called *result type (scheme)* for n .
2. We lift the $\bar{n}(\mathcal{A})$ from data names to $\bar{t}(\mathcal{A})$ for type expressions t :

$$\begin{aligned}
\bar{1}(\mathcal{A}) &= 1 \\
\overline{t_1 \times t_2}(\mathcal{A}) &= (\bar{t}_1(\mathcal{A}) \times \bar{t}_2(\mathcal{A}))
\end{aligned}$$

3. For each constructor c in s there is an *algebra member* $\mathcal{A}.c$ of type $\bar{t}(\mathcal{A}) \rightarrow \bar{n}(\mathcal{A})$, where $c(s) = t \rightarrow n$.

We consider the set of all s -fold algebras as the fold algebra type for the system s .

5.3 Fold functions

Generalised folding for systems s of datatypes can be defined by induction on T . In an application $fold\langle t \rangle \mathcal{A} x$, we require that \mathcal{A} is an s -fold algebra, and x is of type t . The result type of folding is, of course, $\bar{t}(\mathcal{A})$.

$$\begin{aligned} fold\langle 1 \rangle \mathcal{A} () &= () \\ fold\langle t_1 \times t_2 \rangle \mathcal{A} (x_1, x_2) &= (fold\langle t_1 \rangle \mathcal{A} x_1, fold\langle t_2 \rangle \mathcal{A} x_2) \\ fold\langle n \rangle \mathcal{A} (c x) &= \mathcal{A}.c (fold\langle t \rangle \mathcal{A} x) \text{ where } c(s) = t \rightarrow n \end{aligned}$$

The definition of paramorphic fold functions [Mee92,SF93] and monadic fold functions [Fok94,MJ95] (see also Section 4) requires just a modest elaboration of the scheme above. Although we did not illustrate paramorphisms in this paper, we should mention that the recursion scheme underlying paramorphisms is very desirable for traversals where the structure of subterms needs to be observed. Paramorphisms can be encoded as catamorphisms by a tupling technique, but this is very inconvenient in actual programming.

5.4 Basic algebras

Let us now define the basic fold algebras *idmap*, *crush* and *zero* induced by a system s . For all c :

$$\begin{aligned} idmap.c &= c \\ crush.c &= \lambda x. crush'\langle t \rangle x \text{ where } c(s) = t \rightarrow n \\ zero.c &= \lambda x. mzero \end{aligned}$$

The definition of *idmap* is immediately clear. For *crush*, we need to define a generic function *crush'* which performs crushing for parameters of constructors. The definition of this function (and thereby crushing) assumes a monoid $\langle \alpha, e, \circ \rangle$, where α is a type variable, e denotes the neutral element, and \circ denotes the associative operation:

$$\begin{aligned} crush'\langle 1 \rangle () &= e \\ crush'\langle t_1 \times t_2 \rangle (x_1, x_2) &= (crush'\langle t_1 \rangle x_1) \circ (crush'\langle t_2 \rangle x_2) \\ crush'\langle n \rangle x &= x \text{ for all } n \text{ in } s \end{aligned}$$

For *zero*, we assume a monad with zero, that is a structure $\langle M, return, \gg=, mzero \rangle$.

In Section 3, we introduced the terms *type-preserving* and *type-unifying* to describe the classes of algebras of which respectively *idmap* and *crush* are representatives. We can now characterise these classes by the result types of the algebras. For a type-preserving algebra \mathcal{A} , $\bar{\pi}(\mathcal{A}) = n$ for all n in s . For a type-unifying algebra \mathcal{A} , $\bar{\pi}(\mathcal{A}) = \tau$ for all n in s , i.e., there is common result type τ independent of the type index. The basic fold algebra *zero* (or any algebra of the same type) is not restricted to either of these classes. The result types are of the form $\bar{\pi}(zero) = M \alpha$ (with different α for different n), i.e., the result types for the various n in s are only constrained to be monadic.

5.5 Algebra combinators

Sections 3 and 4 featured a number of operators on fold algebras. Algebra *update* is the most important of these operators. The combinators *unit*, *plus*, and *carried* were introduced for monadic fold algebras. The definitions of these monadic combinators are similar to those for the basic algebras above. The definition of updating is more involved.

If the datatype system s contains the constructor name c , i.e., if $c(s)$ is defined, $\mathcal{A}[c/f]$ denotes the update of an s -algebra \mathcal{A} at c by a function f . Initially, we require the type of f to be equal to the type of $\mathcal{A}.c$. Then, updating can be defined as follows:

$$\mathcal{A}[c/f].c' = \begin{cases} f, & \text{if } c = c' \\ \mathcal{A}.c', & \text{otherwise} \end{cases}$$

It is easy to verify that the resulting structure is indeed a proper s -algebra with $\bar{n}(\mathcal{A}[c/f]) = \bar{n}(\mathcal{A})$ for all n in s . The condition that the type of f is equal to the type of $\mathcal{A}.c$ is not too restrictive in the presence of an operator for type specialisation. We will use $\mathcal{A}[\bar{n}/\tau]$ to denote the instantiation of the result type for n in \mathcal{A} to the type τ . The axiomatisation is omitted for brevity. Type specialisation is allowed under the condition that τ is more specific than $\bar{n}(\mathcal{A})$, i.e., if there is a substitution to replace type variables by type schemes in $\bar{n}(\mathcal{A})$ such that it becomes equal to τ . Recall that in the Haskell model, fold algebra types are parameterized datatypes (record types), algebra updating is record updating, and type specialisation is type parameter instantiation.

5.6 Extensions

So far we have restricted ourselves to closed systems of non-parameterized datatypes. For many purposes this is quite sufficient. In the application areas we have in mind, systems of datatypes are derived from syntax definitions, and the class of systems considered so far covers simple BNF notation. Nonetheless, we will now discuss some possibilities for extending our approach to richer classes of datatypes. As will become apparent, such extensions conjure up a wealth of design choices.

Primitive types The system of datatypes of our running example uses the primitive type `String`. Actually, `String` is not quite primitive in Haskell, but defined as list of `Char`. In fact, for pragmatic reasons one may choose to regard any predefined type as primitive. Our approach can be easily extended to handle primitive types. We extend our grammar as follows:

$$\begin{array}{ll} T ::= \dots \mid P & \text{-- additional form of type expression} \\ TS ::= \dots \mid P & \text{-- maintain } TS \supset T \\ P & \text{-- primitive types} \end{array}$$

The axiomatisation of algebras can be extended to provide result types and algebra members for primitive types. This allows to write updates for primitive types. There is an alternative way to cover primitive types, where the axiomatisation of algebras is not affected. The values of primitive types are just preserved during folding as modelled by the following additional case in the inductive definition of *fold*:

$$\text{fold}\langle p \rangle \mathcal{A} x = x$$

Here, p ranges over P . For values of primitive types, *fold* acts like the identity. In Haskell, this is done by having instances of the fold function for primitive types, or as in Example 1, where `fold` simply does not recurse into `String`.

Parameterized datatypes Covering systems of parameterized datatypes is more challenging. Let us stick to uniform recursion of parameters in the sense of regular datatypes. From an application perspective, such an extension allows us to cover extended BNF notation including optionals (maybe type), iteration (lists), nested alternatives (binary sums, *Either*). Note that nested concatenation is already covered by the products of our basic approach.

Let us first extend our grammar to cope with regular datatypes. The syntactical domain S is extended by a form for definitions of regular datatypes, and a form of type expression is added to represent the application of parameterized regular datatypes.

$$\begin{array}{ll} S ::= \dots \mid R = F & \text{-- definition of regular datatypes} \\ T ::= \dots \mid R@T & \text{-- application of regular datatypes} \\ TS ::= \dots \mid R@T & \text{-- maintain } TS \supset T \\ R & \text{-- names of regular datatypes} \\ F & \text{-- regular datatypes (functors)} \end{array}$$

We assume that F is the syntactical domain for regular datatypes (or their functors).

Parameterized datatypes can be handled in essentially the same manner as non-parameterized ones, i.e., by defining additional result types and algebra members for the fold algebra. However,

this extension is not straightforward. The types of the algebra members get more involved. To uniformly handle all instantiations of a particular parameterized datatype in a single algebra member, such members ultimately need to be polytypic functions themselves. Furthermore, it should be possible to enforce specific behaviour for particular applications of a regular datatype.

As for primitive types, there is also a way to cope with parameterized datatypes that does not affect the axiomatisation of algebras. Parameterized datatypes are folded in a homogeneous way based on the polytypic map function (*pmap* in [JJ97]). Consequently, the inductive definition of *fold* is extended as follows:

$$\mathit{fold}\langle r@t \rangle A x = \mathit{pmap} (\mathit{fold}\langle t \rangle A) x$$

Here, r ranges over R . This approach is much easier to formalize. But it is restricted in the sense that updating can not be performed for (constructors of) regular datatypes.

Nested and function types An elaboration to cover nested (rather than just regular) datatypes [BM98,BP99] is not needed for our intended application areas. Nestedness does not commonly occur among large bananas. For similar reasons, function types [MH95] are not considered.

6 Concluding remarks

Contributions The advantages of programming with folds (as opposed to general recursion) are well known. We have presented an elaboration for generalised (monadic) folds on systems of mutually recursive datatypes where a separation is made between fold algebras that capture generic functionality and fold algebra updates that implement problem-specific functionality. This separation provides a combination of generic and specific programming which is crucial to make programming with folds as scalable as possible. We identified a number of particular generic fold algebras as well as some algebra combinators for calculating with monadic folds. Furthermore, we showed that generic definitions can be given of these algebras and combinators, and of the other ingredients for programming with updatable folds.

Our approach is relatively lightweight in two important dimensions: it is conceptually simple, and easy to implement. The first claim can be justified by the argument that, essentially, mastering the concept of generalised folds is sufficient to use the approach. The second claim holds for a generator-based approach, where Haskell functions and datatypes are generated for programming with folds. Our generator took us about 0.1 man years development effort. It is fully operational and can be used for serious case studies as the one reported in [KLV00]. To provide a thorough semantics for our approach and to fully integrate the concepts in a functional language is more ambitious. The integration issue raises the question if such an integration can be done by recasting the approach to some existing generic framework or language such as Charity [CS92], PolyP [JJ97], FISH [Jay99], or Generic Haskell [Hin99]. Such a recasting is not obvious because of the (inherent or current) limitations of the respective languages and approaches. For example, one can not use constructors for selection of algebra members and fold algebra updating. For that purpose, fold algebras need to become first-class citizens in accordance to our axiomatisation.

Related work Polytypic programming [JJ97,Hin00] allows for general recursive type-indexed (or even kind-indexed) functions. On the other hand, we require type-indexed algebra types, i.e., a kind of polytypic datatype definition. To understand the pros and cons of these variations, more research is needed. We should mention one interesting observation, where the restriction to folds pays back in a surprising manner, that is non-monadic traversals (say algebras) can be turned into monadic ones. For general recursive functions, such a migration is inherently subject to program transformation [Läm00], or to semantically restrictive and non-trivial type systems [Fil99].

In [Jon95,SAS99] it is discussed how to program with catamorphisms in Haskell in an (almost) generic way. A generic *cata* is easily defined based on a Haskell class *Functor* whose *fmap* member, however, needs to be instantiated by the programmer for *each* datatype. As noted in [SAS99], elaborate coding is to be done to cope with mutually recursive datatypes. A new functor class

Functor_n is needed for each number n of datatypes. This is not a theoretical problem, but a result of Haskell's limited genericity. Note that datatype definitions must be written as functors in order to fit into this scheme. On the positive side, this allows for modularisation of the datatypes, algebras, and instances of *Functor*.

The tension between genericity and specificity is a recurring theme. Strategies [VBT98,Vis00] have been proposed for term rewriting so that separation is possible of generic phenomena (such as traversal schemes and reduction) and specific ones (one-step rewrite rules). However, the approach is untyped. It is a topic of ongoing research to define strategy operators in a functional and typed setting. A similar duality can be found in recent work on generic functional programming, particularly in [Hin99]. There, so-called ad-hoc definitions are proposed to adapt the uniform way generic values are defined, when specific behaviour for a particular datatype is desirable.

Future work We consider the elaborate treatment of the modularisation of traversals as future work. One dimension of modularity is to cope with incomplete or extensible systems of datatypes. From a functional programming language perspective, concepts like subtyping [AC91] or extensible records [Gas98] seem useful in that context. Another dimension of modularity is the separation of (computational or semantic) aspects involved in traversals. In [MPV99], an aspect-oriented approach to the modularisation of attribute grammars, which can be conceived as traversals in a well-understood sense, is discussed. The approach is based on an encoding in functional programming also relying on extensible records.

Acknowledgment

The authors are indebted to three anonymous referees of WGP'2000 and to Patrik Jansson for invaluable comments and suggestions for the revision of the paper. Ralf Lämmel received partial support from the Netherlands Organization for Scientific Research (NWO) under the *Generation of Program Transformation Systems* project.

References

- [AC91] R.M. Amadio and L. Cardelli. Subtyping recursive types. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 104–118, Orlando, FL, USA, January 1991. ACM Press.
- [BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed lambda -programs on term algebras. *Theoretical Computer Science*, 39(2-3):135–153, August 1985.
- [BM98] R. Bird and L. Meertens. Nested datatypes. In *4th International Conference on Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 52–67. Springer-Verlag, 1998.
- [BP99] R. Bird and R. Paterson. Generalised folds for nested datatypes. *Formal Aspects of Computing*, 11(2):200–222, 1999.
- [BSV00] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36(2–3):209–266, 2000.
- [CC90] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.
- [CS92] R. Cockett and D. Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, 1992.
- [Fil99] A. Filinski. Representing layered monads. In *Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 175–188, New York, N.Y., January 1999. ACM.
- [Fok92] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992.
- [Fok94] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.
- [FSS92] L. Fegaras, T. Sheard, and D. Stemple. Uniform Traversal Combinators: Definition, Use and Properties. In D. Kapur, editor, *Proc. 11th Intl. Conf. on Automated Deduction (CADE-11)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 148–162, Saratoga Springs, NY, USA, June 1992. Springer-Verlag.

- [Gas98] B.R. Gaster. *Records, variants, and qualified types*. PhD thesis, Department of Computer Science, University of Nottingham, 1998.
- [Hin99] R. Hinze. A generic programming extension for Haskell. In Erik Meijer, editor, *Proceedings of the 3rd Haskell Workshop, Paris, France*, September 1999. Technical report, Universiteit Utrecht, UU-CS-1999-28.
- [Hin00] R. Hinze. A new approach to generic functional programming. In Thomas W. Reps, editor, *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21*, pages 119–132, January 2000.
- [Jay99] C.B. Jay. Programming in FISH. *International Journal on Software Tools for Technology Transfer*, 2:307–315, 1999.
- [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.
- [JM95] J. Jeuring and E. Meijer, editors. *Advanced Functional Programming*, volume 925 of *LNCS*. Springer-Verlag, 1995.
- [Jon95] M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Jeuring and Meijer [JM95], pages 97–136.
- [Jon99] M.P. Jones. Type Classes and Functional Dependencies, 1999. <http://www.cse.ogi.edu/~mpj/>.
- [KLV00] J. Kort, R. Lämmel, and J. Visser. Functional Transformation Systems. Draft; available at <http://www.cwi.nl/~ralf/>, March 2000.
- [Läm00] R. Lämmel. Reuse by Program Transformation. In Greg Michaelson and Phil Trinder, editors, *Functional Programming Trends 1999*. Intellect, 2000. to appear.
- [MBJ99] E Moggi, Bellè, and C.B. Jay. Monads, shapely functors and traversals. In M. Hoffman, Pavlović, and P. Rosolini, editors, *Proceedings of the Eighth Conference on Category Theory and Computer Science (CTCS'99)*, volume 24 of *Electronic Lecture Notes in Computer Science*, pages 265–286. Elsevier, 1999.
- [Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
- [Mee96] L. Meertens. Calculate polytypically! *Lecture Notes in Computer Science*, 1140:1–16, 1996.
- [MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire. In *Proc. FPCA '91*, volume 523 of *LNCS*. Springer-Verlag, 1991.
- [MH95] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Intl. Conf. on Functional Programming Languages and Computer Architecture, FPCA '95, La Jolla, San Diego, CA, USA, 25-28 June 1995*, pages 324–333. ACM Press, New York, 1995.
- [MJ95] E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In Jeuring and Meijer [JM95], pages 228–266.
- [MPV99] O. de Moor, S. Peyton-Jones, and E. Van Wyk. Aspect-oriented Compilers. In *Proceedings of GCSE '99 (Erfurt, Germany)*, LNCS. Springer-Verlag, 1999.
- [SAS99] S. Doaitse Swierstra, P. R. A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In Swierstra et al. [SHO99], pages 150–206.
- [SF93] T. Sheard and L. Fegaras. A Fold for All Seasons. In *FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, Copenhagen, Denmark, June 1993. ACM Press. ISBN 0-89791-595-X.
- [She91] T. Sheard. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.
- [SHO99] S.D. Swierstra, P.R. Henriques, and J.N. Oliveira, editors. *Advanced Functional Programming, Third International School, AFP '98*, volume 1608 of *LNCS*, Braga, Portugal, September 1999. Springer-Verlag.
- [VBT98] E. Visser, Z. Benaïssa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *International Conference on Functional Programming (ICFP'98), Baltimore, Maryland. ACM SIGPLAN*, pages 13–26, September 1998.
- [Vis00] E. Visser. Language Independent Traversals for Program Transformation, July 2000. in these proceedings.