Modular Reifiable Matching

A List-of-Functors Approach to Two-Level Types

Bruno C. d. S. Oliveira
University of Hong Kong
bruno@cs.hku.hk

Shin-Cheng Mu
Academia Sinica
scm@iis.sinica.edu.tw

Shu-Hung You
National Taiwan University
suhorngcsie@gmail.com

Abstract

This paper presents Modular Reifiable Matching (**MRM**): a new approach to two level types using a *fixpoint of list-of-functors* representation. **MRM** allows the modular definition of datatypes and functions by pattern matching, using a style similar to the widely popular Datatypes à la Carte (DTC) approach. However, unlike DTC, **MRM** uses a fixpoint of list-of-functors approach to two-level types. This approach has advantages that help with various aspects of extensibility, modularity and reuse. Firstly, modular pattern matching definitions are collected using a list of matches that is fully reifiable. This allows for extensible pattern matching definitions to be easily reused/inherited, and particular matches to be overridden. Such flexibility is used, among other things, to implement *extensible generic traversals*. Secondly, the subtyping relation between lists of functors is quite simple, does not require backtracking, and is easy to model in languages like Haskell. **MRM** is implemented as a Haskell library, and its use and applicability are illustrated through various examples in the paper.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures

Keywords Modular Datatypes, Subtyping

1. Introduction

Pattern matching and *algebraic datatypes* are defining features of languages like Haskell or ML. Pattern matching is useful because it allows for an easy way to do case analysis of complex structures. Moreover definitions using pattern matching can often be reasoned using techniques such as structural induction.

Unfortunately, definitions using pattern matching can be prone to code duplication, lack of extensibility, or boilerplate code. To counter some of these limitations approaches such as *two-level types* [26] have been proposed in the past. The essential idea of two-level types is to separate the definition of a datatype into two parts. Firstly, there is a datatype that represents (part of) the structure that the programmer is interested on. For example, if the goal is to represent (simplified) arithmetic expressions, one could write:

```
data ArithF x = Lit Int | Add x x
```

In this case ArithF abstracts the recursive occurrences of the datatype away, using a type parameter x. Note that, because all occurrences of x are in positive (or argument) positions, ArithF is trivially a functor. Secondly, a datatype Fix:

```
data Fix f = In {out :: f (Fix f)}
```

is applied to an arbitrary functor f to give a type-level *fixpoint* of that functor. Therefore, the datatype of arithmetic expressions is recovered by applying Fix to ArithF.

One big benefit of the two-level types approach is that it does not require datatypes to be monolithically defined. As popularized by the *Datatypes à la Carte* (DTC) [27] approach, it is possible to define modular parts of a datatype and later compose them using co-products (or sums) of functors:

```
data (⊕) f g x = Inl (f x) | Inr (g x)
```

For example, creating a datatype for abstract syntax of arithmetic and boolean expressions could be done by combining two independently defined functors: the ArithF functor above and another LogicF functor with the boolean related expressions. The fixpoint of the co-product of the two functors (Fix (ArithF ⊕ LogicF)) would then represent the abstract syntax datatype for a simple language of arithmetic and boolean expressions.

A final ingredient in the two-level types approach is how to modularly define pattern matching definitions themselves. The common approach, used for example by DTC, is to use type classes [28]. The basic idea is that for each functor there is a corresponding type class instance that defines (part of) the operation for the overall datatype.

Unfortunately there are some wrinkles with the existing two-level types approach, which prevent wider applicability. A well-known technical problem [3, 23] is that the natural subtyping relation on functors cannot be easily defined in languages like Haskell. For example, both of the following should be valid instances of the subtyping relation on functors:

$$g \prec (f \oplus g) \oplus h \qquad\qquad g \prec f \oplus (g \oplus h)$$

However, due to the Haskell encoding of the subtyping relation and limitations of type classes, approaches such as DTC only accept the second instance. Another problem arises from the use of type class instances to capture modular definitions. For certain modular operations, it would be interesting to define the new operation by simply reusing the code for another operation and *overriding* some of the cases. A concrete example would be a *substitution* function for expressions of a programming language. For most parts of the abstract syntax, the substitution function behaves like an *identity traversal*: a traversal that walks the structure but simply rebuilds it in the output. The only cases where substitution differs from the identity traversal is for binders and variables. So, instead of defining substitution from scratch, an alternative approach would be to have an identity traversal and simply override the cases for binders and variables. However this kind of reuse is not easy to achieve with typical type class approaches.

Contributions This paper presents Modular Reifiable Matching (**MRM**), a new approach to two level types using a *fixpoint of list-of-functors* representation. The list-of-functors representation to two-level types has important advantages that help with various aspects of extensibility, modularity and reuse. Importantly **MRM** naturally addresses the wrinkles of the existing two-level types approach described above.

In **MRM** functors are combined with a type-level list-of-functors, instead of using co-products. This allows more precise kinds when working with functors. In particular, unlike approaches using co-products, it is possible to distinguish between *membership* ($\in$) and *subtyping* ($\prec$) of a list-of-functors. Moreover, the subtyping and membership relations can be encoded easily in languages like Haskell as datatypes, which can be reified and traversed to generate operations. This allows **MRM** to rely less on type classes.

In **MRM** it is easy to reuse and override parts of existing operations. Instead of type class instances, **MRM** uses Matches: a special list type that collects parts of definitions by pattern matching. Similarly to the use of type class instances, Matches allows for modularity, since new matches can be added to an existing list of matches. However, unlike type class instances, matches in **MRM** are also reifiable and first class: it is possible to define functions that iterate over a list of matches, or pass a list of matches as arguments to functions. Using this it is easy to *inherit*, *reuse* and *override* matches of existing operations. In particular, it is possible to define highly generic *extensible queries* and *transformations* (a generic identity traversal), and reuse those operations to define traversals that only implement some cases differently.

The use and applicability of **MRM** is shown through several examples and case studies. The paper illustrates different practical scenarios which show how to: *implement extensible and modular definitions*; *encode subtyping of algebraic datatypes and pattern matching definitions for reuse*; *model inheritance and overriding of operations*. Furthermore, two case studies show concrete uses of **MRM**. The first shows how to use **MRM** to define a library of language components. Besides various extensible functions such as evaluation and pretty printing, the library of language components includes two extensible generic traversals, *substitution* and *free variables*, for dealing with binders. Moreover it shows how to modularly implement *constant function elimination*. A second case study shows a simple library of effect handlers modelled in **MRM** as transformations, where the only interesting cases are the operations of the effects being handled.

In summary, the contributions of this paper are:

- **A new Approach to Two-Level Types:** We introduce a new representation for two-level types based on a list-of-functors.
- **An Encoding of Subtyping for Datatypes and Operations:** We show how to encode subtyping of *closed* algebraic datatypes, and corresponding operations.
- **Mechanisms for Inheritance and Overriding of Operations:** We show how to inherit and override matches. This is used, among other things, to define *extensible generic traversals*.
- **Implementation and Case Studies:** The implementation of **MRM** is available online[1]. Furthermore we have two larger case studies: a library of language components; and a library for effect handlers.

[1] http://www.iis.sinica.edu.tw/~scm/2015/mrm

Fixpoint of a list of functors, injection and projection:

```
data Fix (fs :: [* → *]) where ...
inn :: (Functor f, f ∈ fs) ⇒ f (Fix fs) → Fix fs
prj :: (fs ≺ fs, f ∈ fs)    ⇒ Fix fs → Maybe (f (Fix fs))
```

First-class Matches:

```
data Matches (fs :: [* → *]) (a :: *) (b :: *) where ...
Void :: Matches '[] a b
(:::) :: Functor f ⇒
         (f a → b) → (Matches fs a b) → Matches (f ': fs) a b
```

Operations on Matches and Fix:

```
fold  :: Matches fs a a → Fix fs → a
match :: Matches fs (Fix fs) b → Fix fs → b
```

Figure 1. Basic API for **MRM**.

2. An Overview of Modular Reifiable Matching

This section provides an overview of **MRM**. We first introduce the key concepts of the new two-level types approach. Then **MRM** is illustrated using three different application scenarios: datatype extensibility; datatype subtyping; and extensible generic traversals. This section introduces **MRM** from a user point of view. Implementation details of **MRM** are introduced in later sections.

2.1 Two-Level Types in MRM

MRM follows an approach based on two-level types to represent datatypes. Each functor represents part of the overall datatype. For example, the following functor:

```
data ArithF x = Num Int | Add x x deriving Functor
```

describes the shape of a datatype representing arithmetic expressions. The type parameter x abstracts over the recursive occurrences of the datatype. As in the usual two-level types approach, the definition of a datatype requires a second part: a type-level fixpoint that replaces all the occurrences of x by a recursive datatype reference. However **MRM** uses a fixpoint of a *list of functors* instead of the traditional fixpoint of a functor. For example, the datatype of arithmetic expressions is obtained as follows:

```
type Arith = Fix '[ArithF]
```

Here Fix is a type that takes a (type-level) list of functors. The list of functors states all the functors (in this case just one) that contribute constructors for the datatype. The kind of Fix, as well as the types and kinds of other important datatypes and operations in **MRM**, are shown in Figure 1. Note that the use of type-level lists is possible due to *datatype promotion* [31]: a recent feature of GHC Haskell. With datatype promotion it is possible (among other things) to use the syntax for value-level lists at the type-level.

Folding One way to define functions in **MRM** is using a f-algebra and a fold operation taking that algebra. For example, the following code shows how to define a pretty printing operation for arithmetic expressions:

```
ppArithAlg :: ArithF String → String
ppArithAlg (Num x)     = show x
ppArithAlg (Add e₁ e₂) = e₁ ++ "+" ++ e₂

ppArith₁ :: Arith → String
ppArith₁ = fold (ppArithAlg ::: Void)
```

The function ppArithAlg is an f-algebra (where f is ArithF). The **MRM** library provides a fold operator that takes an fs-algebra (encoded as a value of type Matches fs a a) and Fix fs arguments. For each functor $f \in fs$ there must be a corresponding f-algebra. The operator ::: is used to add one algebra to a list-of-algebras; and the data constructor Void denotes an empty list of algebras. In the case of $\mathsf{ppArith}_1$ only one algebra (ppArithAlg) is needed.

Matching In **MRM** it is also possible to define functions using general recursion. For example, here is an alternative version of pretty printing using general recursion instead of a fold:

```
ppArithMatch :: ArithF Arith → String
ppArithMatch (Num x)      = show x
ppArithMatch (Add e1 e2) =
  ppArith2 e1 ++ "+" ++ ppArith2 e2

ppArith2 :: Arith → String
ppArith2 = match (ppArithMatch ::: Void)
```

The match function, also part of **MRM**, allows defining a function by pattern matching. In contrast to fold, match replaces all parameter positions in the functor by values of type Arith. Thus a recursive call has to be done explicitly on those arguments.

The *pattern synonyms* and *view patterns* extensions of GHC Haskell further allows integrating **MRM** with Haskell's built-in pattern matching:

```
pattern NumP x      ← (prj → Just (Num x))
pattern AddP e1 e2 ← (prj → Just (Add e1 e2))

ppArith3 :: Arith → String
ppArith3 (NumP x)     = show x
ppArith3 (AddP e1 e2) = ppArith3 e1 ++ "+" ++ ppArith3 e2
```

For $f \in fs$, the function prj tries to project its input to f (Fix fs) and returns Nothing if failed. Two new patterns NumP x and AppP e_1 e_2 are introduced as pattern synonyms of the corresponding projection.

Matches ***as a List of (Generalized) Algebras*** The Matches datatype is inspired by f-algebras. In the traditional formulation of f-algebras as a sum-of-products

$$f\ a \to a \cong (f_1\ a\ +\ \ldots\ +\ f_n\ a) \to a$$

a functor f is built from smaller functors $f_1\ \ldots\ f_n$. Each smaller functor is typically a product type. The functors are assembled together using sums. An alternative way to represent f-algebras is to use the type-theoretic isomorphism $s + t \to a \cong (s \to a) \times (t \to a)$, leading to the following isomorphic representation:

$$f\ a \to a \cong (f_1\ a \to a)\ \times\ \ldots\ \times\ (f_n\ a \to a)$$

This alternative representation of f-algebras is already quite close in spirit to Matches. However Matches is more general in that it represents a function of type $f\ a \to b$, instead of $f\ a \to a$:

$$f\ a \to b \cong (f_1\ a \to b)\ \times\ \ldots\ \times\ (f_n\ a \to b)$$

Such *generalized* f-algebra type turns out to support many of the same operations as f-algebras. Moreover it allows expressing generally recursive operations quite easily. Therefore Matches is built on top of generalized f-algebras instead of standard f-algebras:

$$\mathsf{Matches}\ [f_1, \ldots, f_n]\ a\ b \cong (f_1\ a \to b)\ \times\ \ldots\ \times\ (f_n\ a \to b)$$

In other words Matches is just a list of generalized algebras with corresponding functors $f_1\ \ldots\ f_n$. Each generalized algebra contributes a partial *match* to the overall pattern matching definition.

2.2 Extensibility and Modularity

One scenario where **MRM** is useful is in the development of extensible and modular datatypes and functions. Similarly to approaches such as Datatypes à la Carte, **MRM** provides the mechanisms that make such modular definitions possible. For example, suppose that the task at hand is to develop modular components for interpreters.

Evaluation of Arithmetic Expressions Section 2.1 already provided the abstract syntax for arithmetic expressions (ArithF). To interpret an expression, we first define a type of values:

```
data VF = N Int | B Bool | F
```

The three cases denote respectively a number, a boolean result, and a failure. Note that here we are not trying to support extensible values as well, to make the example easier to follow. Evaluation of an arithmetic expression is defined as an ArithF-algebra:

```
evArith :: ArithF VF → VF
evArith (Num n)             = N n
evArith (Add (N x) (N y)) = N (x + y)
evArith _                   = F
```

Boolean Expressions Modular abstract syntax and an interpretation function for boolean expressions is added similarly:

```
data LogicF x = Bol Bool | If x x x deriving Functor

evLogic :: LogicF VF → VF
evLogic (Bol n)         = B n
evLogic (If (B b) x y) = if b then x else y
evLogic _               = F
```

The datatype LogicF is a functor that represents the basic abstract syntax for boolean expressions. The evLogic function is a LogicF-algebra that describes the semantics of boolean expressions.

Generic Smart Constructors For convenience, we may define the following smart constructors:

```
num :: ArithF ∈ fs ⇒ Int → Fix fs
num     = inn ∘ Num

add :: ArithF ∈ fs ⇒ Fix fs → Fix fs → Fix fs
add x y = inn (Add x y)

bol :: LogicF ∈ fs ⇒ Bool → Fix fs
bol     = inn ∘ Bol

iif :: LogicF ∈ fs ⇒ Fix fs → Fix fs → Fix fs → Fix fs
iif x y z = inn (If x y z)
```

Here the function inn, a part of the basic API for **MRM**, is used to lift constructors of the functor ArithF into the type Fix fs. Note that the type of inn allows lifting an arbitrary functor f into a value of type Fix fs as long as f is one of the functors in the list fs ($f \in fs$).

A Language with Arithmetic and Boolean Expressions A language supporting arithmetic and boolean expression can be built from the modular components. The first step is to define the type of expressions in the language:

```
type Expr = Fix '[ArithF, LogicF]
```

The type Expr denotes expressions that may have integer and boolean literals, an addition operator, and if-conditionals.

The evaluation function for this language is defined as a fold using the combined algebras:

```
eval :: Expr → VF
eval = fold (evArith ::: evLogic ::: Void)
```

This function can be used with the smart constructors to evaluate a concrete expression:

```
test = eval (iif (bol True) (add (num 3) (num 4)) (num 5))
```

Automatic Generation of Matches **MRM** also supports automatic generation of Matches, similarly to how DTC automatically generates algebras. Figure 2 shows a class Eval, which generates an algebra where the carrier type is VF. There is a base instance for

```
class Eval fs where
  evalAlg :: Matches fs VF VF
instance Eval ('[]) where
  evalAlg = Void
instance Eval fs ⇒ Eval (ArithF ': fs) where
  evalAlg = evArith ::: evalAlg
instance Eval fs ⇒ Eval (LogicF ': fs) where
  evalAlg = evLogic ::: evalAlg
```

Figure 2. Automatic generation of Matches using type classes.

'[], which constructs an empty match using Void. For each functor that supports evaluation, there is also a corresponding instance that adds the algebra to the list of algebras.

A generic evaluation function geval is then defined using evalAlg. This function works for any list of functors fs, where for each functor there is a corresponding instance of Eval.

```
geval :: Eval fs ⇒ Fix fs → VF
geval = fold evalAlg
```

2.3 Subtyping and Reusability

A different scenario where **MRM** is useful is when the programming task in hand involves a set of *closed*, but related datatypes. We want to emphasize here that extensibility is not the important aspect: in such applications, there is usually no need for extensibility. However it is still important to achieve reuse across the datatypes and operations. As pointed out by several researchers the absence of subtyping among algebraic datatypes can lead to significant code duplication [1, 8]. For example, when developing a compiler for a language it is often the case that there is a source language, as well as multiple intermediate languages. Often these languages are closely related and are subsets of each other. However many operations (such as pretty printing) have to be defined for each of these languages, even though there is obviously a lot of repetition!

MRM comes with a set of functions that allow us to exploit the natural subtyping relationships between datatypes. Two of these functions are:

```
subFix :: fs ≺ gs ⇒ Fix fs → Fix gs
subOp :: fs ≺ gs ⇒ (Fix gs → c) → Fix fs → c
```

The function subFix converts between a datatype where the list of functors fs is a subset of another list of functors gs. Conversely, the function subOp converts an operation on a larger datatype into an operation on a smaller datatype. These two functions allow for significant reuse of code. For example, consider the datatypes Arith and Expr. Note that Arith only supports arithmetic expressions, whereas Expr also supports boolean expressions.

Reuse of Constructors and Values If conventional algebraic datatypes are used to model abstract syntax, different constructors are needed for numeric values and addition in the Arith and Expr types. However, in the encoding of Arith and Expr, only a single set of constructors is needed. Moreover expressions of type Arith are compatible with expressions of type Expr:

```
e₁ :: Arith
e₁ = add (num 4) (num 5)

e₂ :: Expr
e₂ = subFix e₁
```

In e_2, subFix is used to coerce e_1 into a value of type Expr.

Reuse of Operations Another form of reuse is that of operations. If Arith and Expr are modelled with conventional algebraic datatypes, then two different evaluation functions are needed to evaluate expressions of the two types. However, in **MRM** it suffices to define evaluation for the larger type of expressions (in this case Expr). Because of the subtyping relation on operations on datatypes, it is possible to reuse the operation on smaller expressions. For example, it suffices to define evaluation of Expr values:

```
evalExpr :: Expr → VF
evalExpr = match ((λcase
  Num n      → N n
  Add e₁ e₂ →
    case (evalExpr e₁, evalExpr e₂) of
      (N x, N y) → N (x + y)
      _          → F) ::: (λcase
  Bol b      → B b
  If e₁ e₂ e₃ →
    case evalExpr e₁ of
      B b → if b then evalExpr e₂ else evalExpr e₃
      _   → F) ::: Void)
```

Evaluation on Arith values can be recovered by applying the subtyping coercion function subOp to evalExpr:

```
evalArith :: Arith → VF
evalArith = subOp evalExpr
```

The definition of evalExpr deserves some additional explanation. That definition uses general recursion and the match operator. Since the list-of-functors contains two functors (ArithF and LogicF), a list of two matches (one for each functor) is needed. To make the definition as close as possible to a definition using Haskell pattern matching the matches are inlined and separated by the ::: operator. Additionally we use a simple *lambda-case* (λ**case**) syntactic sugar, which allows us to do anonymous case analysis. The lambda-case sugar was recently added to the GHC compiler. It is possible to have the same definition without lambda-case, but the definition is slightly more cumbersome. Lambda-case turns out to be a handy form of syntactic sugar to program such kinds of definitions.

Finally note that an advantage of not having to care about extensibility is that definitions and corresponding types can be significantly simpler. If extensibility is also a goal, then evalExpr needs to be split into various modular parts, and each part requires more complex types.

2.4 Extensible Generic Traversals

A final application of **MRM** is to do extensible generic traversals. **MRM** supports two main kinds of generic traversals: *queries* and *transformations*. Both of these traversals are extensible in the sense that it is possible to create new traversals from existing ones. We use extensible queries as an example of generic traversals here. Both queries and transformations are discussed in detail in Section 5. The simpler interface for generic queries is:

```
queryAlg :: (Traversables fs, Monoid a) ⇒ Matches fs a a
```

The definition queryAlg produces a collection of matches of type f a → a. This definition is generic in the sense that it is defined for any list-of-functors fs, where all f ∈ fs are instances of the Traversable class[2]. Using solely queryAlg results in a query that merely returns the identity of the monoid:

```
emptyQuery :: (Monoid a, Traversables fs) ⇒ Fix fs → a
emptyQuery = fold queryAlg
```

[2] Note that all the functors defined in this paper are also Traversable. We omit routine Traversable instances from the paper.

Free Variables A more interesting example is a generic function that, for any language that supports the binding constructs,

```
data Var x = Var String
data Let x = Let String x x
```

computes the list of free variables of the given term. When we encounter a Var x, we simply return x in a singleton list. In the case Let x vs ws, x has to be removed from the free variables of the body (ws).

```
fVarsAlg :: (Traversables fs, Var ∈ fs, Let ∈ fs) ⇒
             Matches fs [String] [String]
fVarsAlg =
   (λ(Var x)          → [x]) >::
   (λ(Let x vs ws) → vs ‘union‘ (ws ⟍ x)) >:: queryAlg
xs ⟍ x = filter (not ∘ (x ==)) xs
```

For other cases we fall back to the default case specified by queryAlg. Since lists are monoids, the free variables of the subcomponents are combined using (++). If there is no subcomponent, the result is simply the identity [] of the monoid. Note that we use an operator >:: to *override* an existing match in the structure:

```
(>::) :: f ∈ fs ⇒ (f a → b) → Matches fs a b → Matches fs a b
```

With this operator the two binding cases defined in fVarsAlg override the default cases provided by queryAlg. Finally, the list of free variables may be collected by folding the algebra:

```
freeVars₁ = fold fVarsAlg
```

Different Semantics and New Language Extensions A generic free variables function, which accounts for extensible syntax, must necessarily be extensible. For example, adding new language extensions with binders requires extending free variables with new cases. Moreover, sometimes a language may reuse the same abstract syntax, but have a different interpretation for that syntax. In that case the old interpretation must be overridden. In the following example we illustrate both cases, and make the necessary adaptations by extending fVarsAlg.

```
data Lam x = Lam String x

fVarsAlgExt :: (Traversables fs, Lam ∈ fs, Var ∈ fs,
                   Let ∈ fs) ⇒ Matches fs [String] [String]
fVarsAlgExt =
   (λ(Let x vs ws) → (vs ‘union‘ ws) ⟍ x) >::
   (λ(Lam x vs)    → vs ⟍ x) >:: fVarsAlg
```

Here, the idea is that a lambda binder construct is included. Additionally **let** expressions are interpreted differently: instead of a simple **let** we now assume that **let** is interpreted as letrec (which changes how free variables are computed). In order to account for these changes we need two things: 1) the Let interpretation for free variables must be overridden; and 2) we need a new case for the lambda construct. However, the case for variables is still inherited from fVarsAlg.

A concrete language that uses the new version of free variables can be created as follows:

```
type Exp = Fix '[Var, Let, Lam, ArithF, LogicF]


freeVars₂ :: Exp → [String]
freeVars₂ = fold fVarsAlgExt
```

Note that we never had to define free variables for the cases in ArithF and LogicF. Those cases were inherited from queryAlg.

3. Basic Infrastructure

In this section we introduce the basic infrastructure supporting **MRM**, including fixpoints, match, fold, and paramorphism.

3.1 Fixpoint of List-of-Functors

Fixpoints in **MRM** are represented using the following datatype:

```
data Fix (fs :: [∗ → ∗]) where
  In :: Functor f ⇒ Elem f fs → f (Fix fs) → Fix fs
```

In each node of a tree of type Fix fs we may use any f which is a member of fs. The constructor of Fix thus takes evidence that f is indeed a member of fs (Elem f fs) as an argument. The type Elem f fs is defined as follows:

```
data Elem (f :: ∗ → ∗) (fs :: [∗ → ∗]) where
  Here  :: Elem f (f ': fs)
  There :: Elem f fs → Elem f (g ': fs)
```

The constructor Here witnesses that f occurs in f ': fs. If xs is evidence that f occurs in fs, There xs is evidence that f occurs in g ': fs. One can also see Elem f fs as an unary index into fs. For example, There (There (There Here)) :: Elem f fs states that f occurs at the third position of fs, counting from the zeroth position.

It is cumbersome having to provide the evidence each time. Fortunately, the evidence can be generated by a type class with two (overlapping) instances: f is certainly a member of f ': fs, and if f is in fs, it is in g ': fs as well.

```
class f ∈ fs where
  witness :: Elem f fs
instance f ∈ (f ': fs) where
  witness = Here
instance (f ∈ fs) ⇒ f ∈ (g ': fs) where
  witness = There witness
```

The smart fixpoint constructor is then defined as:

```
inn :: (f ∈ fs, Functor f) ⇒ f (Fix fs) → Fix fs
inn = In witness
```

The* Matches *Datatype As explained in Section 2.1, Matches fs a b is the type of a list of functions f a → b, for each f in fs, where Void is the empty list, while (:::) attaches a function f a → b to the left of a list:

```
data Matches (fs :: [∗ → ∗]) (a :: ∗) (b :: ∗) where
  Void :: Matches '[] a b
  (:::) :: Functor f ⇒ (f a → b) → Matches fs a b
          → Matches (f ': fs) a b
```

Using Elem f fs as an index, we can extract the corresponding function f a → b in Matches fs a b:

```
extractAt :: Elem f fs → Matches fs a b → (f a → b)
extractAt Here           (f ::: _)  = f
extractAt (There pos) (_ ::: fs) = extractAt pos fs
```

A list Matches fs (Fix fs) b can be applied to a fixpoint Fix fs by the function match:

```
match :: Matches fs (Fix fs) b → Fix fs → b
match fs (In pos xs) = extractAt pos fs xs
```

3.2 Fold for List-of-Functors Datatypes

Many functions in this paper are folds. To fold over a value of type Fix fs, we need a list of f-algebras, one for each f in fs, which is a special case of matches:

$$\textbf{type}\ \mathsf{Algebras\ fs\ a} = \mathsf{Matches\ fs\ a\ a}$$

The fold for $\mathsf{Fix\ fs}$ can be defined as below:

```
fold :: Algebras fs a → Fix fs → a
fold ks (In pos xs) = extractAt pos ks (fmap (fold ks) xs)
```

In each step of the computation, fold extracts from ks the algebra designated by the index pos. Note that fmap is also instantiated to the functor f designated by pos.

Like ordinary folds, our fold possesses a universal property (where, for clarity, we let $\mathsf{fmap_f}$ denote the f instance of fmap):[3]

Theorem 3.1 (Universal Property)**.** *For all* $\mathsf{h :: Fix\ fs \to a}$ *we have* $\mathsf{h = fold\ ks}$ *if and only if*

$$(\forall \mathsf{pos :: Elem\ f\ fs.}\ \ \mathsf{h \circ In\ pos = extract\ pos\ ks \circ fmap_f\ h})$$

from which a fusion law directly follows:

Theorem 3.2 (Fusion)**.** *Given* $\mathsf{ks :: Algebras\ fs\ a}$, $\mathsf{h :: a \to b}$, *and* $\mathsf{gs :: Algebras\ fs\ b}$, *we have* $\mathsf{h \circ fold\ ks = fold\ gs}$ *if*

$$(\forall \mathsf{pos :: Elem\ f\ fs.}\ \ \mathsf{h \circ extract\ pos\ ks =}\ \mathsf{extract\ pos\ gs \circ fmap_f\ h})$$

3.3 Simple Operations on Matches, and Paramorphism

Matches are not arrows, since one cannot come up with a general definition of arr. However, many other arrow operators do turn out to be useful on matches. To begin with, pure functions can be composed before and after matches:

```
(≪^) :: Matches fs b c → (a → b) → Matches fs a c
Void ≪^ g       = Void
(h ::: hs) ≪^ g = (h ∘ fmap g) ::: (hs ≪^ g)

(^≪) :: (b → c) → Matches fs a b → Matches fs a c
g ^≪ Void       = Void
g ^≪ (h ::: hs) = (g ∘ h) ::: (g ^≪ hs)
```

Having $(^\wedge\!\!\ll)$ actually makes $\mathsf{Matches\ \ fs\ \ a}$ a functor, if we let $\mathsf{fmap} = (^\wedge\!\!\ll)$. Given two list of matches having the same input type, one can construct their "split":

```
(&&&) :: Matches fs a b → Matches fs a c
          → Matches fs a (b, c)
Void       &&& Void       = Void
(h ::: hs) &&& (k ::: ks) = (λx → (h x, k x)) ::: (hs &&& ks)
```

Paramorphism Not all functions are folds. All primitive recursive functions, however, can be implemented by a pattern called a *paramorphism* [21]. A paramorphism differs from a fold in that, in each step, we can use the recursively computed result as well as the original subtrees. Paramorphisms can be implemented by a fold that returns the desired result together with a copy of the input.

```
para :: Matches fs (a, Fix fs) a → Fix fs → a
para ks = fst ∘ fold (ks &&& (inns ≪^ snd))
```

A step of the computation in a paramorphism can be modelled by $\mathsf{ks :: Matches\ \ fs\ \ (a, Fix\ \ fs)\ \ a}$: each subtree has been folded to $\mathsf{(a, Fix\ fs)}$, a result and a copy of the input, from which we should compute some result of type a. To re-assemble the original input, we use $\mathsf{inns} \ll^\wedge \mathsf{snd}$. Here inns is the initial algebra for Fix fs:

```
inns :: Algebras fs (Fix fs)
```

The definition of inns is a special case of transAlg, which we present in Section 4.2. The split ks &&& $(\mathsf{inns} \ll^\wedge \mathsf{snd})$ thus has type $\mathsf{Algebra\ fs\ (a, Fix\ fs)}$, which we can use in a fold.

[3] An Agda proof of the universal property for polynomial base functors is included in the distributed code.

4. Subtyping in Matches and Fixpoints

The matches and fixpoints introduced in the previous section are useful to encode modular and reusable function definitions and datatypes. These constructions naturally lead to questions about the relationship between matches or fixpoints which use related sets of functors. When a match or fixpoint uses a subset of functors of another match or fixpoint, we may expect that some subtyping relation exists. This section discusses those subtyping relationships and formalizes them as part of **MRM**.

4.1 Subtyping, Intuitively

Although many functional programming languages do not support subtyping, it is interesting to consider what subtyping relationships are expected to hold for algebraic datatypes. Consider the following Haskell pseudo-code:

```
data Exp = Lit Int | Add Exp Exp
data ExpExt extends Exp  = Mul Exp Exp
```

The first definition is just a standard Haskell datatype declaration. The second definition is not valid Haskell, but expresses the intention of declaring the datatype ExpExt as an extension of Exp. In other words ExpExt should have all the constructors of Exp and, additionally, Mul.

Subtyping Between Datatypes Lets consider what happens with values of the datatypes. If we have an expression:

```
e₁ :: Exp
e₁ = Add (Lit 4) (Lit 5)
```

then it should be clear that it is always safe to use this expression as a value of ExtExp as well.

```
e₂ :: ExpExt
e₂ = e₁
```

After all ExpExt contains all the constructors of Exp. This indicates that Exp is a subtype of ExpExt. More generally datatype extensions (i.e. with additional constructors) become *supertypes* of the extended datatypes.

Subtyping Between Operations Now consider the following definition, which performs evaluation ExpExt, respectively:

```
evalExpExt :: ExpExt → Int
evalExpExt (Lit x)       = x
evalExpExt (Add e₁ e₂) = evalExpExt e₁ + evalExpExt e₂
evalExpExt (Mul e₁ e₂) = evalExpExt e₁ ∗ evalExpExt e₂
```

ExpExt extends Exp, and evalExpExt deals with all the cases of Exp already. Thus it should be clear that it is safe to also apply the function to values of type Exp:

```
evalExp :: Exp → Int
evalExp = evalExpExt
```

This hints for another subtyping relation between functions on related datatypes: functions of type $\mathsf{ExpExt \to A}$ are *subtypes* of functions with type $\mathsf{Exp \to A}$ (for some type A). More generally functions, whose input are of the type of the datatype extension, become subtypes of functions where the input type is the extended datatype. This is a consequence of the contravariant nature of functions with respect to subtyping.

Subtyping Relations and Extension The subtyping relationships can be summarized as follows, with respect to extension:

(extension)	ExpExt	**extends**	Exp
(datatype subtyping)	Exp	$\prec$	ExpExt
(operation subtyping)	$\mathsf{ExpExt \to A}$	$\prec$	$\mathsf{Exp \to A}$

Note that the subtyping relation on datatypes follows the inverse direction of the extension: ExpExt is an extension and a *supertype* of Exp. This is unlike the subtyping relation between operations or the usual subtyping relation between classes in mainstream OO languages, which follow the same direction of the extension. Technically speaking, the subtyping relation on operations is said to be *covariant* (it follows the same direction as extension); whereas the subtyping relation on datatypes is *contravariant* (it follows the opposite direction of extension).

4.2 Modelling Subtyping

The intuitive notions of subtyping discussed in Section 4.1 can be modelled precisely in **MRM**. Essentially an operation defined by pattern matching corresponds to a set of matches, whereas a datatype corresponds to a fixpoint of functors.

Subset of Lists of Functors The first step is to model a subset relation between two lists of functors:

```
data Sub (fs :: [∗ → ∗]) (gs :: [∗ → ∗]) where
  SNil  :: Sub '[] gs
  SCons :: (Functor f) ⇒
              Elem f gs → Sub fs′ gs → Sub (f ': fs′) gs
```

The basic idea is that a list fs represents a subset of another list gs if every member of fs is also a member of gs. This is expressed inductively with two cases: (SNil) if fs is empty, then it clearly is a subset of any list gs; (SCons) for f ': fs′ to represent a subset of gs, f must be a member of gs and fs′ must represent a subset of gs. Evidence for the subset relation can be built automatically using a type class fs ≺ gs, if we can generate evidence Sub fs gs. The two cases mentioned above are respectively taken care of by two instance declarations.

```
class fs ≺ gs where
  srep :: Sub fs gs
instance '[] ≺ gs
  where srep = SNil
instance (Functor f, f ∈ gs, fs ≺ gs) ⇒ (f ': fs) ≺ gs
  where srep = SCons witness srep
```

Importantly note that the relation fs ≺ gs does not require backtracking and can be defined in a straightforward way using Haskell type classes.

Fixpoint Subtyping The subtyping relation between datatypes can be materialized as a coercion function:

```
subFix :: (fs ≺ gs) ⇒ Fix fs → Fix gs
subFix = fold transAlg
```

This function converts a value of type Fix fs to a less restrictive type Fix gs, which can be done because every functor allowed by fs is also allowed by gs. In terms of implementation, subFix, expressed as a fold, takes an algebra transAlg (to be read transformation algebra), defined in terms of the auxiliary transAlg′:

```
transAlg′ :: Sub fs gs → Algebras fs (Fix gs)
transAlg′ SNil              = Void
transAlg′ (SCons pos xs) = In pos ::: transAlg′ xs
```

Note that In pos has type f (Fix gs) → Fix gs for some f. For In to type check, we have to provide an evidence that f is a member of gs, for which we have pos, extracted from the evidence of Sub fs gs.

```
transAlg :: (fs ≺ gs) ⇒ Algebras fs (Fix gs)
transAlg = transAlg′ srep
```

The function transAlg basically generates a sequence of In, each supplied an index. The function inns introduced in Section 3.3 is a special case when fs = gs.

Operation Subtyping Using the subtyping relation of fixpoints, it is easy to define subtyping of operations as a coercion function:

```
subOp :: (fs ≺ gs) ⇒ (Fix gs → c) → Fix fs → c
subOp f = f ∘ subFix
```

Matches Subtyping Finally, because in **MRM** matches are first-class, we have an additional subtyping relation between matches. With the subset relation Sub, the subtyping relation between matches can be materialized as a coercion function:

```
subMatch′ :: Sub fs gs → Matches gs r a → Matches fs r a
subMatch′ SNil              _  = Void
subMatch′ (SCons pos xs) as =
  extractAt pos as ::: subMatch′ xs as
subMatch :: (fs ≺ gs) ⇒ Matches gs r a → Matches fs r a
subMatch = subMatch′ srep
```

The function subMatch′ expresses the fact that we can always convert from a larger set of matches to a (smaller) subset of these matches. The function subMatch provides a variant of subMatch′ which takes the evidence for the subset relationship implicitly.

Summary of Subtyping Relations The following table shows a summary of subtyping relations with respect to the subset relation:

(**subset relation**)	fs	≺	gs
(**fixpoint subtyping**)	Fix fs	≺	Fix gs
(**operation subtyping**)	Fix gs → A	≺	Fix fs → A
(**matches subtyping**)	Matches gs a b	≺	Matches fs a b

Note that in this case, the subtyping relation on fixpoints is covariant with respect to the subset relation. In contrast, both operation and matches subtyping are contravariant with respect to the subset relation.

5. Queries and Transformations

In this section we demonstrate how various operations on datatypes can be implemented in our framework. Like previous work on generic traversals [15] we start with two kinds of operations: a *transformation* traverses its input and returns a result of the same type, possibly altered, while a *query* yields a result of a fixed type.

In our running examples, we will use the following base functors, intended to denote λ terms,

```
type VName = String
data Var x  = Var VName    data Lam x = Lam VName x
data App x = App x x        data Let x  = Let VName x x
```

with smart constructors

```
var  :: Var ∈ fs  ⇒ VName → Fix fs
app :: App ∈ fs ⇒ Fix fs → Fix fs → Fix fs
lam :: Lam ∈ fs ⇒ VName → Fix fs → Fix fs
lett :: Let ∈ fs  ⇒ VName → Fix fs → Fix fs → Fix fs
```

Routine instance derivations (Functor, etc) and definitions of the smart constructors are omitted.

5.1 Generic Transformations: Reifying Subtyping

The function subFix, introduced in Section 4.2, can be seen as a simple transformation. At the level of values, subFix merely copies each constructor. The type, due to transAlg, states that the output can be the fixpoint of some gs that is a superset of fs — gs must accommodate at least all the functors in fs, and could allow more.

In fact, transAlg acts as the "default case" of transformations, which keeps the input unchanged. More complex transformations can be formed by extending transAlg, in a style similar to Lämmel et al. [19]. For example, the following function rename′ traverses through the input, renames each variable name s by s ++ "_", while leaving other constructors untouched.

```
rename′ :: (fs ≺ gs, Var ∈ gs) ⇒ Fix (Var ′: fs) → Fix gs
rename′ = fold ((λ(Var s) → var (s ++ "_")) ::: transAlg)
```

It is instructive examining the type of rename′. The constraint Var ∈ gs comes from the use of var — the output could be the fixpoint of any gs, as long as it allows Var and all functors in fs. The input must be the fixpoint of Var ′: fs, because a match expecting Var is appended the front of transAlg. The function can be applied to any fs and gs meeting the constraints, leaving room for further extension, until they are instantiated at the site of application.

It is more flexible if Var may appear in places other than the head. The call overrideAt pos k ks replaces, by k, the match in ks designated by pos:

```
overrideAt :: Elem f fs → (f a → b) →
                    Matches fs a b → Matches fs a b
overrideAt Here         g (_ ::: fs) = g ::: fs
overrideAt (There pos) g (f ::: fs) = f ::: overrideAt pos g fs
```

We may then define an operator (>::) that overrides a match by f a → b, as long as f is provably in fs:

```
(>::) :: f ∈ fs ⇒ (f a → b) →
                         Matches fs a b → Matches fs a b
(>::) = overrideAt witness
```

Using (>::) we can define a rename that accepts any Fix fs as long as fs allows Var:

```
rename :: (fs ≺ gs, Var ∈ gs, Var ∈ fs) ⇒ Fix fs → Fix gs
rename = fold ((λ(Var s) → var (s ++ "_")) >:: transAlg)
```

In most applications, a function like rename is not supposed to change the type of its input. Indeed, we may let fs = gs, and will do so for examples later.

The prj function introduced in Section 2.1 can also be implemented by overriding a Just match in a list of constant matches generated by constMatch.

```
prj :: (fs ≺ fs, f ∈ fs) ⇒ Fix fs → Maybe (f (Fix fs))
prj = match (Just >:: constMatch Nothing)
```

Narrowing Transformations So far, it appears that the output of a transformation is in general less restrictive than its input. In rename :: ... Fix fs → Fix gs, we have fs ≺ gs. Indeed, we can always cast Fix fs to a less restrictive Fix gs using subFix.

However, there are also ways to constraint the output to be more restrictive than the input. Consider the task, which we call "desugaring", of translating a language with λ-abstraction, application, and Let-binding into a core language without Let. The only case we need to explicitly handle is Let:

```
desugarAlg = (λ(Let x e₁ e₂) → app (lam x e₂) e₁) :::
                   transAlg
```

The most general type we can give to desugarAlg is

```
desugarAlg :: (fs ≺ gs, App ∈ gs, Lam ∈ gs) ⇒
                  Algebras (Let ′: fs) (Fix gs)
```

since Let is the first component of the matches, and the use of app and lam demands that gs allows App and Lam.

We can, however, let gs = fs and give both desugarAlg and the following desugar a more restrictive type:

```
desugar :: (fs ≺ fs, App ∈ fs, Lam ∈ fs) ⇒
              Fix (Let ′: fs) → Fix fs
desugar = fold desugarAlg
```

It is clearer that the input type allows Let, while the functor is removed from the head of the list in the output. The output is thus "narrower" than the input. We will see more applications of such narrowing transformations in Section 6.

Note that fs ≺ fs is not a redundant constraint. It cannot be discharged not only because the type class mechanism of Haskell is not expressive enough to determine that (≺) is reflexive, but because fs ≺ fs also demands that each element in fs is a Functor.

Transformation in a Paramorphism We now look at a more complex transformation: define subst such that subst v t e substitutes each free occurrence of variable v in term e for term t. The occurrences of v in Lam w e, if v = w, are bound and must not be substituted.[4]

The call subst v t explicitly handles two cases: Var and Lam, and thus only requires its input to contain these two cases. In the Var w case we perform substitution if v = w. In the Lam w e′ case when v = w, however, e′ is already substituted in the recursive call. We need a copy of the original, unsubstituted body. The function subst v t is therefore a paramorphism.

The main work is performed in substMatch defined below, a list of matches whose input type is (Fix fs, Fix fs), the first component being the substituted term, while the second the original term. Notice, in the Lam case, that the recursive part now contains a pair (e′, e), where e′ is the substituted body while e is the original one.

```
substMatch :: (fs ≺ fs, Var ∈ fs, Lam ∈ fs) ⇒
  String → Fix fs → Matches fs (Fix fs, Fix fs) (Fix fs)
substMatch v t =
  (λ(Var w) → if (v == w) then t else var w) >::
  (λ(Lam w (e′, e)) →
    if (v == w) then lam w e else lam w e′) >::
  transAlg ≪^ fst
```

The default cases in substMatch are constructed by transAlg ≪^ fst, which extracts the first components computed from the subtrees, that is, the substituted terms, and assembles them. We can use substMatch in para, introduced in Section 3.3:

```
subst :: (fs ≺ fs, Var ∈ fs, Lam ∈ fs) ⇒
         String → Fix fs → Fix fs → Fix fs
subst v t = para (substMatch v t)
```

Thus subst is a substitution operator that works for any term that contains Var and Lam, with other cases being default cases. Note also that substMatch is extensible, just as the matches for free variables presented in Section 2.4. In the case Lam w (e′, e), due to lazy evaluation, the substituted body e′ is only evaluated when necessary. Alternatively, one may use a "Mendler-style" paramorphism:

```
mpara :: (∀r.(r → a, r → Fix gs) → Matches gs r a)
         → Fix gs → a
mpara ks (In pos xs) = extractAt pos (ks (mpara ks, id)) xs
```

where ks may choose to makes recursive calls only when necessary.

5.2 Queries: Reifying Traversable Functors

Queries also traverse the input data. Unlike transformations, they collect information of a fixed type during the traversal. One thus has to specify how the information collected from the subtrees is combined.

The Haskell type class Monoid models a monoid with an identity element ι and an associative binary operator ($\oplus$). If a type constructor f is in class Traversable, we have the following method:[5]

[4] Being only a short example, we do not deal with name capturing.

[5] The method actually belongs to class Foldable, but all Traversable types are Foldable.

```
foldMap :: Monoid m ⇒ (a → m) → f a → m
```

The function foldMap k traverses the input f-structure, converts components of type a to m, and combines them using (⊕). To perform a query on Fix fs, we demand that the result type is a monoid, and each f in fs is Traversable. We define a datatype TList fs, an evidence that every member in fs is in the class Traversable, and let the class Traversables generate such evidences:

```
data TList (fs :: [∗ → ∗]) where
  TNil   :: TList '[]
  TCons :: Traversable f ⇒ TList fs → TList (f ': fs)

class Traversables (fs :: [∗ → ∗]) where trep :: TList fs
instance Traversables '[] where trep = TNil
instance (Traversable f, Traversables fs) ⇒
          Traversables (f ': fs) where
  trep = TCons trep
```

The function qMMatch generates a list of matches by applying a function a → b to all the values, and combines them using foldMap, if b is a monoid:

```
qMMatch :: Monoid b ⇒
  TList fs → (a → b) → Matches fs a b
qMMatch TNil k         = Void
qMMatch (TCons xs) k = foldMap k ::: qMMatch xs k
```

Like transformations, we start with a basic query algebra that does nothing, to be extended for each application. The list of algebras queryAlg assumes that the carrier a is a monoid and collects them using (⊕).

```
queryAlg :: (Monoid a, Traversables fs) ⇒ Algebras fs a
queryAlg = qMMatch trep id
```

More usage of queries has been demonstrated in Section 2.4.

5.3 Monadic Matchings

For more complex computation, we may need to carry a state, produce output, or raise exception during the traversal. That is, we need the transformation to yield monadic results.

MRM was designed from scratch to allow reification of the matches in the Matches structure. We can lift pure matches Matches fs a b to monadic matches of type Matches fs (m a) (m b). It is important to note that monadification was previously presented as a program transformation [11], whereas in **MRM** it is actually definable as a function.

A functor f in the class Traversable provides a method

```
sequence :: Monad m ⇒ f (m a) → m (f a)
```

which we may use to combine monadic results of the substructures. The function mlift′ goes through a list of pure matches and replaces each k :: f a → b by liftM k ∘ sequence, of type f (m a) → m b:

```
mlift′ :: Monad m ⇒ TList fs → Matches fs a b
                              → Matches fs (m a) (m b)
mlift′ TNil         Void      = Void
mlift′ (TCons ts) (k ::: ks) =
  (liftM k ∘ sequence) ::: mlift′ ts ks

mlift :: (Monad m, Traversables fs) ⇒
         Matches fs a b → Matches fs (m a) (m b)
mlift = mlift′ trep
```

The default transformation transAlg can be lifted to a list of monadic matches that re-assembles its input, while performing monadic actions:

```
transMAlg :: (Monad m, Traversables fs, fs ≺ gs) ⇒
             Algebras fs (m (Fix gs))
transMAlg = mlift transAlg
```

Lifting Pure Algebras Recall that in Section 2.2 we defined a small language of expressions, with two evaluation algebras:

```
evArith :: ArithF VF → VF
evLogic :: LogicF VF → VF
```

Suppose now that we want to extend the language with variables and **let**-binders. An environment is needed to keep track of variables and their values, and a *reader monad* comes in handy here to help managing the environment:

```
type Env = [(VName, VF)]
evVar :: MonadReader Env m ⇒ Var (m VF) → m VF
evVar (Var x) =
  do env ← ask
     return (maybe F id (lookup x env))
evLet :: MonadReader Env m ⇒ Let (m VF) → m VF
evLet (Let x e₁ e₂) =
  do v₁ ← e₁
     local ((x, v₁):) e₂
```

The evaluation algebra itself is unsurprising. Lookup errors are handled using the failure value (F). In the evaluation of **let** binders the environment is locally extended with the result of evaluating e_1. Existing algebras evArith and evLogic may then simply be converted to work with the monadic algebras:

```
eval :: Fix '[Var, Let, ArithF, LogicF] → VF
eval e = runReader (fold alg e) [] where
  alg = evVar ::: evLet ::: mlift (evArith ::: evLogic ::: Void)
```

Some final words, before we close this section, on the expressiveness of our queries/transformations library: the situation in **MRM** is not different from that in other combinator-based libraries. The provided combinators are fairly general and can deal with a lot of cases, but there are programs that cannot be handled with the combinators. However users can also write their own combinators and extend **MRM** and support additional applications.

6. Case Studies

In this section we look at two larger examples. The first uses a mixture of transformation, query, folding, and matching. The second is inspired by monadic effect handling.

6.1 Constant Function Elimination

We have developed a library of language components, some of which (e.g. substitution and free variables) have been presented. Here we discuss a more involved transformation adopted from Lämmel et al. [19].

A term (λx → e) is a constant function if x does not occur free in e. In our simple language, occurrences of (λx → e_1) e_2, where (λx → e_1) is a constant function, can be replaced by e_1 without changing its semantics. Our task is to perform such replacements.

```
pattern LamP x body ← (prj → Just (Lam x body))
pattern AppP e₁ e₂    ← (prj → Just (App e₁ e₂))
cfe1 (AppP (LamP x e₁) e₂)
    | not (x ∈ freeVars e₁) = e₁
cfe1 (In pos xs)              = In pos (fmap cfe1 xs)
```

Here we create two pattern synonyms as in Section 2.1. If x occurs free in body in the AppP-LamP case (freeVars is $\mathsf{freeVars}_2$ with a more general type), then lam x e_1 is a constant function and we can extract its body. Otherwise we recursively apply cfe1 to the input.

Let the input type be Fix fs. One can see that cfe1 demands that fs allows App and Lam, while the call to freeVars further demands that fs allows Var and Let. Indeed, cfe1 can be assigned the following type:

```
cfe1 :: (Traversables fs, Var ∈ fs, App ∈ fs, Lam ∈ fs,
         Let ∈ fs, fs ≺ fs) ⇒ Fix fs → Fix fs
```

Optimizing cfe The function cfe1 calls freeVars once in every node, as was done by Lämmel et al. [19]. To avoid repeatedly traversing the tree, we can cache the free variables in the tree.

We define a functor Anno a to annotate datatypes. The function annoAlg converts every algebra ks :: Algebras fs a to an algebra that returns the result of ks, together with an annotated tree, where each subtree is ornamented with values computed from it. Function annotate for trees is like scanr for lists:

```
data Anno a x = Anno a x
annoAlg :: ((Anno a) ∈ gs) ⇒
  Sub fs gs → Algebras fs a → Algebras fs (Fix gs, a)
annotate :: ((Anno a) ∈ gs, fs ≺ gs) ⇒
              Algebras fs a → Fix fs → Fix gs
annotate ks = fst ∘ fold (annoAlg srep ks)
```

Conversely, unannotate is a narrowing transformation that removes the annotations:

```
unannotate :: (fs ≺ fs) ⇒ Fix (Anno a ': fs) → Fix fs
```

A more efficient cfe2 may then be implemented in three stages:

```
cfe2 :: Fix '[Var, App, Lam, Let] → Fix '[Var, App, Lam, Let]
cfe2 = unannotate ∘ cfe1' ∘ annotate fVarsAlgExt
```

Firstly, annotate fVarsAlgExt annotates every subtree of the input tree with its free variables (fVarsAlgExt is the algebra used by freeVars_2 in Section 5.2). Secondly, cfe1′ is like cfe1, but uses stored information rather than calling freeVars. Finally, unannotate removes the annotations.

6.2 Effect Handling

Various approaches to implement *algebraic effects* have recently become a popular alternative to monad transformers. Aiming at dealing with the shortcomings of the latter, algebraic effects allow more flexible interaction among effects, and allow programs using different subsets of effects to be composed more easily. Inspired by the approach of Wu et al. [30], we present effect handling as a collection of narrowing transformations. In this approach, an effectful program is represented as an abstract syntax tree that can be interpreted by layers of interpreters.

Free Monads Recall that the *free monad* of a functor f is given by Free a = a + f (Free a). In **MRM**, we may define

```
data Pure a x = Pure a
```

such that the free monad of a list of functors fs is obtained by Free fs a = Fix (Pure a ': fs). We wrap it in a **newtype** definition, with a deconstructor getFix and a smart constructor free:

```
newtype Free fs a = Free {getFix :: Fix (Pure a ': fs)}
free :: (Functor f, f ∈ fs) ⇒ f (Free fs a) → Free fs a
```

An effectful program is represented by the free monad of fs, the effects allowed in the program. The following are some of the notable effects that can be represented this way:

```
data State s  x = Get (s → x) | Put s x
data Nondet   x = Or x x
data Except e x = Throw e
```

A program allowing the effect State s, for example, is an abstract syntax tree in which operators Get and Put may appear, while a program with effect Except is allowed to use the operator Throw.

Monad Instance These effectful operators are combined using return and (≫=). We have to show that they can be defined for any fs, that is, Free fs is a monad. The method return is simply Pure lifted to Free:

```
instance (fs ≺ fs) ⇒ Monad (Free fs) where
  return = Free ∘ inn ∘ Pure
```

To define (≫=), note that the only places where values of type a appear in a syntax tree Free fs a are those leaves labelled by Pure. Therefore, m ≫= k should traverse m, find those leaves, and apply k, which can be done in a fold:

```
(Free p) ≫= f = fold ((λ(Pure x) → f x) :::
                       transFree) p
```

where transFree :: (fs ≺ gs) ⇒ Algebras fs (Free gs a) is transAlg lifted to Free fs.

Now that Free fs is a monad we can use the **do**-notation to develop programs. The following program (which is an abstract syntax tree) raises an exception if the state is negative, otherwise decrements its state non-deterministically by one or two:

```
dec = do n :: Int ← get
         if n < 2 then throw "too small"
         else do {i ← choose 1 2; put (n − i)}
```

In this program get, put, choose, etc. are respectively smart constructors for Get, Put, Or. The syntax tree has the following type, and can be freely combined with other syntax trees that allow these effects.

```
dec :: ((State Int) ∈ fs, Nondet ∈ fs,
       (Except String) ∈ fs, fs ≺ fs) ⇒ Free fs ()
```

Effect Handling Having built the syntax trees, we have to define how to evaluate them. Effects are discharged by layers of *effect handlers*, each of them discharging one effect while changing the result type of the program:

```
runNondet :: (fs ≺ fs) ⇒ Free (Nondet ': fs) a → Free fs [a]
runExcept :: (fs ≺ fs) ⇒ Free (Except e ': fs) a →
                           Free fs (Either e a)
runState  :: (fs ≺ fs) ⇒ Free (State s ': fs) a →
                           (s → Free fs (a, s))
```

The handler runNondet, for example, removes Nondet from the head of the list of effects, while returning a program whose result type is [a]. Similarly, runState takes a program with the effect State s, and returns a function from a state to a program that yields type (a, s), with State s removed from the head position. After all the effects are discharged, we are left with Free '[] a: a tree containing only one single node, whose value we can be extracted by a single match.

```
runPure :: Free '[] a → a
runPure = match ((λ(Pure x) → x) ::: Void) ∘ getFix
```

A program of type Fix '[State Int, Nondet], for example, can be executed by:

```
runPure ∘ runNondet ∘ flip runState 0
```

It is known that different order of effect handling induces different semantics. A program Fix '[State Int, Nondet] reduces to s → Free '[] [(a, s)], each nondeterministic branch having its own state. In contrast a program Fix '[Nondet, State Int] yields s → Free '[] ([a], s), where the states are threaded among branches. The order of effects in the type, however, can be freely altered by subFix before the program is executed.

Let us look at how effect handlers are defined. Each handler deals with two cases: Pure, and its own effect. The hander runNondet, for example, returns [x] in the Pure case. In the Or case, both mx and my have type Free fs [a]. We thus use liftM2 (++) to combine their results:

```
runNondet = fold ((λ(Pure x)      → return [x]) :::
                  (λ(Or mx my) → liftM2 (++) mx my) :::
                  transFree) ∘ getFix
```

The handler runState is slightly more complex. In cases for Get and Put, g has type s → s → Free fs (a, s), while k has type s → Free fs (a, s). The aim is to construct a function s → Free fs (a, s) while passing the states around:

```
runState =
  fold ((λcase Pure x  → (λs → return (x, s))) :::
        (λcase Get g    → (λs → g s s)
                  Put s′ k → (λs → k s′)) :::
        algST) ∘ getFix
```

The default cases are handled by another combinator algST :: (fs ≺ fs) ⇒ Algebras fs (s → Free fs (a, s)), which can be defined by generating matches calling fmap ($s). Details omitted. Except e can also be handled in a similar manner, and the catch operator can be implemented using (>::). It traverses the syntax tree, and replaces each Throw e by a call to the exception handler hdlr.

```
catch :: (fs ≺ fs, (Except e) ∈ fs) ⇒
  Free fs a → (e → Free fs a) → Free fs a
catch m hdlr = fold ((λ(Pure x)    → return x) :::
                     (λ(Throw e) → hdlr e) >::
                     transFree) ∘ getFix $ m
```

Plenty of tricky issues about effect handling are not covered here, however. Section 7 discusses some of these issues.

7. Related Work

Throughout the paper we already discussed the relationship with closely related work on two-level types and Datatypes à la Carte. In this section we discuss additional related work.

Improving and Extending DTC Bahr and Hvitved [4] created an extensive library of reusable components using DTC techniques. Similarly to us they present a way to do desugaring (a particular case of transformations) using DTC. However their approach is different from ours: they do not define desugaring as an extension of an identity traversal. Instead desugaring is defined from scratch using type classes (in the usual DTC style). There is one generic default instance that performs the default behaviour, and then there are dedicated instances for the cases do actually do desugaring. They also show how to do generic (non-extensible) queries and transformations. In contrast to our approach, a key difference is that once a query or transformation is defined, it cannot be later extended or overridden with new cases. So their approach would be unsuitable to define modular components for generic free variables or substitution. Furthermore, they did not discuss how to encode subtyping of closed datatypes and operations as we did, and their approach still suffers from the functor subtyping limitations.

Later work by Bahr [3] addresses the limitation of DTC in terms of subtyping. He proposed an alternative implementation of the subtyping relationship using *closed type families* [10] and a number of auxiliary type family relations. This approach retains the use of co-products in the original DTC implementation. However it requires significantly more machinery than the original DTC infrastructure for functor subtyping. In contrast, our approach is to change the way two-level types are represented, instead of trying to stick to the original approach using co-products. As we have shown this leads to easily definable subtyping and membership relations. Morris and Jones [23] show that with *instance chains*, a proposal for a Haskell language extension, it would be possible to properly encode the subtyping relation on functors. Unfortunately instance chains are not yet adopted by mainstream Haskell compilers such as GHC. Using Coq type classes, which support backtracking, it is possible to directly encode the subtyping relation on functors [9].

Two-Level Types Interestingly, in the thesis of Malcolm [20] where he proved the existence of initial algebras for polynomial functors, he actually started with using lists of functors before switching to single functors of co-products. It appears that this representation has since been forgotten in time, perhaps because it was not directly representable on the languages of that time.

A different approach to two-level types was presented by Axelsson [2], who showed that the so-called *AST model* and *symbol domains* can replace, respectively, type-level fixpoints and functors. Similarly to DTC it is possible to have a subtyping relation on symbol domains using co-products. However this approach inherits the limitation of the subtyping relationship on co-products. In contrast **MRM** still uses a type-level fixpoint, but moves away from co-products and uses a list-of-functors representation instead.

First-Class Patterns The Matches datatype in **MRM** allows pattern matching to be treated as first class and in a type-safe way. Rhiger's work on type-safe pattern combinators [25] illustrated how to define first-class type-safe pattern matching combinators. Rhiger's goals were however quite different from ours. He intended to show that it is possible to express pattern matching combinators without relying on datatypes, while being reasonably efficient. **MRM** is focused on extensibility and reification of patterns, which was not explored in Rhiger's work.

Traversals Much work has been spent on generic traversals, whose use was popularized by the "Scrap your Boilerplate" approach [16–18]. In their last paper of the series [18], Lämmel and Peyton Jones discuss the need for *extensible* generic traversals (queries and transformations), on which significantly less work has been done. However, the traversals of Lämmel and Peyton Jones still lack one form of extensibility that **MRM** supports: the extensibility of the datatypes themselves. In order to support modular generic functions to deal with binding constructs (such as free variables and substitution), both forms of extensibility are needed: extensibility of algebraic datatypes is needed to allow new programming language extensions, and extensibility of generic traversals is needed to allow dealing with new binding constructs.

MRM's approach to generic queries and transformations is closely related to Lämmel et al.'s work on fold-algebras [19]. In their work they deal with so-called *large bananas* (big fold-algebras). Among the operations they propose, idmap corresponds to an (identity) transformation, and crush corresponds to a query. Also closely related to **MRM**'s traversals is Bringert and Ranta's [6] compos family of traversals, which uses the Traversable type class to unify and generalize queries and transformations. **MRM**'s use of the Traversable class in generic queries is partly inspired by the compos approach. However, in contrast to **MRM**, the idmap, crush and compos functions are defined manually for each *datatype*, and the datatypes are not extensible.

Subtyping and Extensibility for Algebraic Datatypes Ahn and Sheard [1] proposed a Haskell extension with *shared subtypes*. This extension allows subtyping of algebraic datatypes, and avoids code duplication for similar functions across the datatypes and their subtypes. **MRM** provides similar functionality. Outside of Haskell there have also been proposals for language extensions supporting extensible datatypes. Extensible ML [22] is an extension to the ML language that supports a combination of OO-style features with algebraic datatypes. In Extensible ML it is possible to use datatype variants to simulate classes, and function cases to simulate methods. Moreover, it is also possible to add new cases to existing datatypes and functions, allowing Extensible ML to support an hybrid programming style in-between functional and object-oriented programming. **MRM** supports extensibility of datatypes and functions, but deep OO hierarchies are not supported. Polymorphic variants [12] are a well-known feature of OCaml that allows a form of *structural* variants. The membership constraints in **MRM** are somehow similar to types for polymorphic variants, since they allow expressions which (set of) variants are required by a function. In contrast to all of such approaches, which rely on language extensions, **MRM** is just a library.

Effect Handlers and Algebraic Effects Since the series of papers from Plotkin et al. showing that effects can be combined [13] and handled [24], various approaches have been experimented to implement the idea. Part of the inspiration for the list-of-functors representation comes from various works on effect handlers [5, 14], which use type-level lists (or encodings of lists) to keep track of the effects used in a program. We generalize this idea and apply it to the representation of two-level types. Of course, as our case study illustrates, it is possible to use the more general representation of two-level types to neatly implement a library for effects. Our implementation of the effects library is most influenced by Wu et al. [30]. One insight that we got from our work is that effect handling closely related to the work on generic traversals: effect handlers are just narrowing transformations in our work. As far as we know this connection has not been made explicit before. It could be that this connection brings further insights on how to deal with challenges that effect handlers pose. This will be an interesting avenue for future work.

8. Conclusion

We have presented Modular Reifiable Matching, an alternative approach to Two-Level Types. Instead of a single functor consisting of sums, we collect the summands into a list of functors. To match its fixpoint, we use a list of matches. This gives us two advantages. Firstly, we can precisely model the subtyping relation, represent the subtyping evidence as a datatype, and inspect the evidence to generate new matches. Secondly, unlike type class instances, matches are first class and can be manipulated. We may easily override, reuse, combine, extend, and monadify existing operations.

Extensible transformations and queries are therefore constructed by extending or overriding a basic list of matches that deals with default cases. A match for substitution, for example, is defined by extending matches of the identity traversal. Only interesting cases need to be explicitly mentioned, and the matches so constructed remain extensible. The **MRM** framework is demonstrated with various examples. For the future we will further develop our case studies. We are planning to have a robust library of language components, including components that deal with binder boilerplate. It will be interesting to develop a library of binding components that deals with sophisticated binders, which have been previously targeted by dedicated libraries [7, 29]. We are also planning to develop a more powerful library for effect handling.

References

[1] K. Y. Ahn and T. Sheard. Shared subtypes: subtyping recursive parametrized algebraic data types. In *Symposium on Haskell*, pages 75–86, 2008.

[2] E. Axelsson. A generic abstract syntax model for embedded languages. In *ICFP '12*, pages 323–334. ACM, 2012.

[3] P. Bahr. Composing and decomposing data types: a closed type families implementation of data types à la carte. In *WGP '14*, pages 71–82. ACM, 2014.

[4] P. Bahr and T. Hvitved. Compositional data types. In *Workshop on Generic Programming*, WGP '11, pages 83–94, 2011.

[5] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP '13*, pages 133–144. ACM, 2013.

[6] B. Bringert and A. Ranta. A pattern for almost compositional functions. In *ICFP '06*, pages 216–226. ACM, 2006.

[7] J. Cheney. Scrap your nameplate. In *ICFP '05*, 2005.

[8] B. C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *ECOOP '09*, July 2009.

[9] B. Delaware, B. C. d. S. Oliveira, and T. Schrijvers. Meta-theory à la carte. In *POPL '13*, 2013.

[10] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *POPL '14*, 2014.

[11] M. Erwig and D. Ren. Monadification of functional programs. *Science of Computer Programming*, 52(1-3):101–129, 2004.

[12] J. Garrigue. Programming with polymorphic variants. *ACM SIGPLAN Workshop on ML*, 1998.

[13] M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

[14] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Symposium on Haskell*, pages 59–70. ACM, 2013.

[15] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *TLDI '03*, pages 26–37. ACM, 2003.

[16] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *TLDI'03*, 2003.

[17] R. Lämmel and S. Peyton Jones. Scrap more boilerplate: Reflection, zips, and generalised casts. In *ICFP '04*, 2004.

[18] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: Extensible generic functions. In *ICFP '05*, pages 204–215. ACM, 2005.

[19] R. Lämmel, J. Visser, and J. Kort. Dealing with large bananas. In *WGP '00*, pages 46–59, 2000. Technical Report, Universiteit Utrecht.

[20] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Groningen University, The Netherlands, 1990.

[21] L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4:413–424, 1992.

[22] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst.*, 26(5), Sept. 2004.

[23] J. G. Morris and M. P. Jones. Instance chains: Type class programming without overlapping instances. In *ICFP '10*, 2010.

[24] G. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP '09*, pages 80–94. Springer-Verlag, 2009.

[25] M. Rhiger. Type-safe pattern combinators. *Journal of Functional Programming*, 19:145–156, 2009.

[26] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587, September 2004.

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

[28] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76. ACM, 1989.

[29] S. Weirich, B. A. Yorgey, and T. Sheard. Binders unbound. In *ICFP '11*, 2011.

[30] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Symposium on Haskell*, pages 1–12. ACM, 2014.

[31] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving haskell a promotion. In *Workshop on Types in Language Design and Implementation*, TLDI '12, 2012.