

Structures on Subcategories in Haskell

Dominic Orchard

University of Cambridge, UK
dominic.orchard@cl.cam.ac.uk

Abstract. Ideas and concepts from category theory have been increasingly adopted in functional programming as a tool for abstraction. For example, many parametric data types are instances of *functors* and *monads*, providing a useful mathematical structuring for programming. However, some parametric data types are restricted to a small set of parameter types in order to provide an efficient implementation e.g. unboxed arrays restricted to primitive types. Such data types do not fit into current mathematical abstractions in Haskell because of these constraints to their polymorphism. This paper discusses how such data types are instances of structures defined over *subcategories*, where Haskell *type classes* define subcategories and *constraint families* allow these subcategories to be indexed by types. This paper describes useful structures on subcategories in Haskell, including *non-endofunctors*, i.e. functors with distinct source and target categories, *relative monads* (introduced by Altenkirch et al.), and *relative comonads* (new in this work), providing the benefits of clear mathematical structuring for programming with efficient data structures.

1 Introduction

Structures from abstract mathematics and category theory have been embraced in functional programming as tools for abstraction, such as *algebras*, *coalgebras*, *functors*, *monads* [22, 23], *monoidal functors* [11], and recently *comonads* [9, 21].

Functors in Haskell are traditionally described by the following *type class*:

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

The *Functor* class describes a functor as a *parametric* data type *f* for which the *fmap* operation is defined. (Note: Haskell has no mechanism for describing or enforcing equational laws, thus these must be checked by hand).

Many data types are instances of *Functor*, for example lists, where *map*::(*a* -> *b*) -> [*a*] -> [*b*] is the standard map operation on lists:

```
instance Functor [] where fmap = map
```

There are however many parametric data types which cannot be made an instance of *Functor*. A well known example is the *Set* data type which has a *map*-like operation of type:

$$Set.map :: (Ord a, Ord b) \Rightarrow (a \rightarrow b) \rightarrow Set a \rightarrow Set b$$

Set.map restricts its element types to members of the *Ord* class because its implementation uses balanced binary trees for efficiency, thus the elements of a set must be comparable [1].

A *Set* instance of *Functor* cannot be defined with $fmap = Set.map$ (the type checker prevents it) as *fmap* does not expect constraints on its type parameters, whilst *Set.map* constrains its type parameters to members of the *Ord* class.

Type classes in Haskell can be interpreted as *subcategories* (noted by [12]). If Haskell is viewed as a category (let's call it **Hask**) with types as *objects* and functions as *morphisms*, then a subcategory has a subset of these types and functions. A type class defines a subcategory whose objects are those types which are instances of the type class.

With this perspective, *Set* is a functor whose domain is restricted to the *Ord* subcategory of **Hask**. The *Functor* class is however defined such that any functor instance is unrestricted in its domain; there are no constraints, thus its domain is the (full) category **Hask**. Thus, we believe *Functor* is a misnomer; *Functor* actually models *endofunctors* whose domain and codomain are **Hask**.

Since *Set* is not an endofunctor (its domain restricted to the *Ord* subcategory of **Hask**) it should not be a surprise that *Set* cannot be made an instance of *Functor*, which describes endofunctors. Thus, the reason *Set* cannot be made an instance of *Functor* is not just a language issue but a *theoretical* issue; not just a type mismatch but a *mathematical* mismatch.

Set is not the only non-endofunctor data type. Another example is the *UArray* data type of unboxed arrays which has a map-like operation:

$$\begin{aligned}
 amap :: (IArray UArray e', IArray UArray e, Ix i) \Rightarrow \\
 (e' \rightarrow e) \rightarrow UArray i e' \rightarrow UArray i e
 \end{aligned}$$

UArray is a functor but the constraints on the type parameters imply it is a functor with a subcategory domain, thus it cannot be an instance of *Functor*.

How then can we define a functor class that allows the subcategory domain to vary per functor? Can a class of non-endofunctors be defined with a parameter for a subcategory domain? This paper uses the *constraint families* extension to Haskell, introduced in the author's previous work, to define such structures.

Constraint families allow type class constraints to be indexed by types [14]. Using a constraint family, a constraint can be varied *per instance* of a class, depending on the instance type. Section 3 provides a recap on constraint families.

Following from the subcategories perspective, and the notation of constraint families, Section 4 shows non-endofunctors and *relative monads* (monads defined on functors that may not be endofunctors [3]) in Haskell. Section 4.3 dualises relative monads to *relative comonads* and shows various examples.

The recent *constraint kinds* extension to the Glasgow Haskell Compiler (GHC) by Bolingbroke [4] extends the capabilities of Haskell's type system such that constraint families can be easily emulated. Section 5 shows the simple application of constraint kinds to provide constraint families in Haskell.

Section 2 begins, introducing the notion of subcategories in Haskell in detail.

2 Subcategories and the Hask Category

Definition 1. A category \mathcal{C} comprises a class of objects $|\mathcal{C}|$, a class of morphisms which map between objects of \mathcal{C} (where $X \rightarrow Y$ denotes a morphism with source object X and target object Y), an associative composition operator for morphisms \circ , and an identity morphism $id_A : A \rightarrow A$ for every object, which is the unit of the composition operator.

The pure semantics of Haskell (with respect to a lack of side effects), and its mathematical style, make it particularly suitable as an (approximate) meta-language for category theory. Haskell may be interpreted as modelling a particular category, let's call it **Hask**, with Haskell types as *objects* and functions as *morphisms*. Composition is provided by standard function composition, and identity morphisms by the polymorphic function $id\ x = x$.

Despite Haskell's lazy semantics and the presence of undefined values, which might threaten to undermine the soundness of applying mathematical laws, equational reasoning with categorical laws in Haskell is palatable and practical [7]. There is some debate as to the exact nature of **Hask**, but certainly it is *cartesian closed* i.e. it has exponential objects (thus functions are also objects) and finite products (tupling). In this paper we need not concern ourselves overly with the particulars of **Hask**; it is sufficient to regard **Hask** as some arbitrary category into which Haskell programs provide definitions¹.

The introduction suggested that type classes in Haskell can be interpreted as describing subcategories of **Hask**. Subcategories are defined formally as such:

Definition 2. For a category \mathcal{C} , a subcategory \mathcal{S} of \mathcal{C} consists of a subclass of the objects of \mathcal{C} and a subclass of the morphisms of \mathcal{C} such that:

- for every morphism $f : X \rightarrow Y$ in \mathcal{S} then $X, Y \in |\mathcal{S}|$
- for every pair of morphisms $f : X \rightarrow Y, g : Y \rightarrow Z$ in \mathcal{S} their composition $g \circ f : X \rightarrow Z$ is in \mathcal{S}
- for every object $X \in |\mathcal{S}|$ the identity morphism $id_X : X \rightarrow X$ is in \mathcal{S} .

An *inclusion functor*, $I : \mathcal{S} \rightarrow \mathcal{C}$ maps objects and morphisms from the subcategory back into its *supercategory*. The subcategory relation will be denoted \sqsubseteq . Above, $\mathcal{S} \sqsubseteq \mathcal{C}$, where \mathcal{C} is the supercategory of \mathcal{S} .

Type Classes as Subcategories As described in the introduction, type classes can be understood as describing subcategories of **Hask**. Consider some class S :

```
class S x where ...
```

S defines a subcategory of **Hask**, where $|S|$ are the types that are instances of S , and the morphisms are functions of type: $(S\ a, S\ b) \Rightarrow a \rightarrow b$ for some a and b . Such a subcategory is in fact a *full subcategory*, which means it contains all the Haskell functions whose source and target types are instances of S .

¹ One might say we view Haskell as an *internal language* of category theory (see [16]).

The inclusion functor $I : S \rightarrow \mathbf{Hask}$ is implicit and ephemeral. Since all types that we can describe and use ultimately belong to \mathbf{Hask} , and not to any other category, any object can be implicitly regarded as being in \mathbf{Hask} , or any morphism as mapping objects in \mathbf{Hask} , thus applying I is redundant, or at least, it is implicit.

Superclass Constraints and Sub-subcategories A declaration of a class may provide constraints over the type parameters of the class, known as *superclass constraints*. For example:

```
class Eq a => Ord a where ...
```

Therefore any type that is an instance of *Ord* must be an instance of *Eq*. Thus, *Eq* is the *superclass* of *Ord*. In the subcategories interpretation, *Ord* is then a subcategory of *Eq*, which is a subcategory of \mathbf{Hask} i.e. $Ord \sqsubseteq Eq \sqsubseteq \mathbf{Hask}$.

Data Types as Object Mappings A parametrically polymorphic data type, with one parameter, e.g.

```
data D a = ...
```

defines a type constructor D of kind $* \rightarrow *$ i.e. taking a type and returning a type. Since the type parameter a is unrestricted in its polymorphism, D can be understood as the *object mapping* $D : |\mathbf{Hask}| \rightarrow |\mathbf{Hask}|$ between objects of \mathbf{Hask} .

GADTs [15] allow the type parameters of a data constructor to be constrained in their polymorphism e.g.:

```
data D a where
  MkD :: S a => a -> D a
```

Here D has kind $* \rightarrow *$, or is an object mapping $D : |\mathbf{Hask}| \rightarrow |\mathbf{Hask}|$. However since there are no other data constructors for D , the object mapping is essentially $D : |S| \rightarrow |\mathbf{Hask}|$ as a D value can only be constructed with a type from S .

3 Constraint Families Recap

Constraint families [14] are analogous to *type families* in Haskell [6, 17].

Type Families allow types to depend on, or be indexed by, other types, thus they provide a simple form of type-function. For example, the following defines a type-level projection function on tuple types via a single-parameter type family:

```
type family Fst t
type instance Fst (a, b) = a
type instance Fst (a, b, c) = a
...
```

The first line declares the name of the type family and its arity. The remaining lines declare instances which define the mappings of the family. Type families are *open* and thus subject to a number of restrictions such that termination can be guaranteed, even in the presence of separate compilation (see [17] for details).

Type families can be *associated* with a type class, where the type parameters of the family (below c) must be at least the type parameters of the class (i.e. must include the type parameters of the class, and possibly further parameters):

```
class Collection  $c$  where
  type Elem  $c$ 
  insert :: Elem  $c \rightarrow c \rightarrow c$ 
```

(when a type family is associated to a class the **family** and **instance** keywords are elided.) Instances of *Collection* must provide an instance of *Elem* e.g.

```
instance Collection ( $[a], [b]$ ) where
  type Elem ( $[a], [b]$ ) = ( $a, b$ )
  ...
```

This instance specifies a collection of a pair of lists where the element type of the collection is a pair of elements.

Constraint Families [14] have a similar syntax and semantics to type families, allowing constraints to depend upon, or be indexed by, types. They may be written separately from a class, or in associated form.

The following example from [14] specifies a polymorphic embedded domain-specific language (in “finally tagless” style [5]), where the language is defined via a class, allowing different evaluation semantics for the language based on some type. An associated constraint family allows each instance to vary its constraints:

```
class Expr  $sem$  where
  constraint Constr  $sem a$ 
  const :: Constr  $sem a \Rightarrow a \rightarrow sem a$ 
  add :: Constr  $sem a \Rightarrow a \rightarrow sem a$ 

data E  $a = E \{ eval :: a \}$ 
instance Expr E where
  constraint Constr E  $a = Num a$ 
  const  $c = E c$ 
  add  $e1 e2 = E \$ eval e1 + eval e2$ 
```

The semantics of terms in the *Expr* language is indexed by the *sem* type. Each instance of *Expr* can have its own specialised constraints via an instance of the *Constr* constraint family.

Usefully, an associated constraint family or type family can have a default instance in the class declaration e.g. *Expr* could have been declared:

```
class Expr  $sem$  where
  constraint Constr  $sem a = ()$ 
  ...
```

where $()$ is the empty constraint. Instances of *Expr* may then omit an instance of the *Constr* constraint family.

4 Structures Parameterised by Subcategories

We will now go on to use constraint families to define abstract categorical structures that have type-indexed constraints, describing subcategories. We begin with functors, for which we will discuss a number of issues.

4.1 Functors

The standard definition of functors in category theory is a map between categories that preserves composition and identities:

Definition 3. For a category \mathcal{C} and category \mathcal{D} , a functor $F : \mathcal{C} \rightarrow \mathcal{D}$ comprises two mappings:

- for all objects, $A \in |\mathcal{C}|$ to $FA \in |\mathcal{D}|$
- for all morphisms, $f : X \rightarrow Y \in \mathcal{C}$ to $Ff : FX \rightarrow FY \in \mathcal{D}$

such that: [F1] $Fid_X = id_{FX}$
 [F2] $F(g \circ f) = Fg \circ Ff$

Endofunctors have the same source and target category i.e. $F : \mathcal{C} \rightarrow \mathcal{C}$.

As seen in the introduction, functors in Haskell are traditionally defined:

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

satisfying analogous laws to those in the mathematical definition:

```
[F1] fmap id = id
[F2] fmap (g ∘ f) = (fmap g) ∘ (fmap f)
```

Parametric data types provide the object mapping $|\mathbf{Hask}| \rightarrow |\mathbf{Hask}|$, and *fmap* completes the definition of a functor by providing the morphism-mapping part.

Taking the type class interpretation of subcategories, instances of *Functor* model endofunctors from $\mathbf{Hask} \rightarrow \mathbf{Hask}$ since there are no class constraints on *a* or *b*. The default restriction of functors to endofunctors is sufficient for defining many Haskell data types as functors. However, some data types are not endofunctors. As discussed, *Set* is one such data type which is not an endofunctor but is a functor $\mathbf{Ord} \rightarrow \mathbf{Hask}$, i.e. from the *Ord* subcategory of \mathbf{Hask} . A type class could be defined to capture all functors with an *Ord* subcategory source:

```
class OrdFunctor f where
  ordfmap :: (Ord a, Ord b) => (a -> b) -> f a -> f b
```

However, defining such a class for every functor needed is tedious and inelegant. Using constraint families, constraints on *a* and *b* can depend on the particular data type for which we are defining a functor, thus the source subcategory can be specified for each instance. Such a class is defined:

```

class Functor f where
  constraint SubCat f a = ()
  fmap :: (SubCat f a, SubCat f b) => (a -> b) -> f a -> f b

```

Given this type class, *Set* can be made an instance of *Functor*:

```

instance Functor Set where
  constraint SubCat Set a = Ord a
  fmap = Set.map

```

Likewise, *UArray* can be made an instance of *Functor*:

```

instance Ix i => Functor (UArray i) where
  constraint SubCat UArray a = IArray UArray a
  fmap = amap

```

It is worth noting that the new definition of the *Functor* class is also backwards compatible with existing instances of *Functor* as it specifies a default empty constraint. Thus, all previous *endofunctors* are valid instances of the new *Functor* class as they need not specify a constraint family instance.

If a type class of endofunctors restricted to **Hask** is required, a definition similar to the original *Functor* type class, perhaps named *EndoFunctor*, could still be provided.

Note that, as an object mapping $Set : |\mathbf{Hask}| \rightarrow |\mathbf{Hask}|$ because the data type itself does not have constraints². As a functor $Set : Ord \rightarrow \mathbf{Hask}$, and as far as the *Set* functor is concerned the object mapping is $|Ord| \rightarrow |\mathbf{Hask}|$.

Contravariance and covariance So far we have kept the target category of *Functor* as **Hask**. This is acceptable since functors are *covariant* in their target, and *contravariant* in their source, with respect to the subcategory relationship. Thus, the target of a functor can always be generalised to its supercategory, in this case **Hask**, and the source of a functor can always be specialised to a further subcategory.

As an example of specialising the source subcategory to a further subcategory, consider a class *Cmp* for which *Ord* is a superclass e.g.

```

class Ord a => Cmp a where ...

```

Therefore $Cmp \sqsubseteq Ord \sqsubseteq Eq \sqsubseteq \mathbf{Hask}$, thus the *Set* instance of *Functor* could be safely specialised to:

```

instance Functor Set where
  constraint SubCat Set a = Cmp a
  fmap = Set.map

```

² It is possible to construct an empty set and a singleton set with elements not in *Ord* by *empty* :: *Set* *a* and *singleton* :: *a* -> *Set* *a* respectively.

and any *Set* value to which *fmap* is applied would still have the *Ord* constraint satisfied, since *Cmp* is a subcategory of *Ord*.

By contravariance, we could not declare that **constraint** *SubCat Set a = Eq a*, as *Eq* is supercategory of *Ord*. The type checker would prevent this erroneous contravariance violation.

4.2 Relative Monads

Monads in Haskell are traditionally defined by the following type class:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

satisfying the following laws, for all $f :: a \rightarrow m\ b$, $g :: b \rightarrow m\ c$:

```
[M1] x >>= return = x
[M2] (return x) >>= f = f x
[M3] x >>= (\lambda x' -> (f x') >>= g) = (x >>= f) >>= g
```

The *Monad* class models a monad in *Kleisli triple form* [10] with object mapping $m : |\mathbf{Hask}| \rightarrow |\mathbf{Hask}|$. The operation ($\gg=$) is often referred to as *bind*.

An alternate, but equivalent definition of monads, requires that m is an endofunctor, as opposed to just an object mapping with the same source and target [10]. Since monads must also be endofunctors, data types that are not endofunctors are also not monads. However, data types which are non-endofunctors may be *relative monads* [2, 3], a generalisation of monads to non-endofunctors. Relative monads are monad-like structures over a functor $J : \mathcal{J} \rightarrow \mathcal{C}$ for some distinct categories \mathcal{J} and \mathcal{C} .

Definition 4. A relative monad *over categories* \mathcal{J} and \mathcal{C} *comprises*:

- a functor $J : \mathcal{J} \rightarrow \mathcal{C}$
- an object mapping $T : |\mathcal{J}| \rightarrow |\mathcal{C}|$
- unit operation: $\eta : JX \rightarrow TX$
- extension operation: $(-)^* : (JX \rightarrow TY) \rightarrow (TX \rightarrow TY)$

with the same monad laws as above (modulo the presence of J in the types, and $(-)^*$ as the prefix version of ($\gg=$)).

Using constraint families, relative monads can be defined in Haskell as:

```
class RMonad t where
  constraint J t a
  unit :: J t x => x -> t x
  extend :: (J t x, J t y) => (x -> t y) -> t x -> t y
```

Here, $\mathcal{J} = J\ t$ and $\mathcal{C} = \mathbf{Hask}$. Thus, the T object mapping of the relative monad is $T : |J\ t| \rightarrow |\mathbf{Hask}|$, which is provided by the parametric data type t . The

functor J of the relative monad is $J : J t \rightarrow \mathbf{Hask}$, which is provided by the *inclusion functor* for the subcategory $J t$ (see Section 4.1), and is implicitly applied to the object x in the signatures of *unit* and *extend* i.e. x is implicitly mapped to its supercategory \mathbf{Hask} .

Set is a relative monad, where the object mapping T is the *Set* type constructor, mapping objects from the *Ord* subcategory to objects of \mathbf{Hask} . The source category of the functor J is defined by the *Set* instance of the J type family, and is the *Ord* subcategory of \mathbf{Hask} . The *Set* data type thus has the following instance of *RMonad*:

```
instance RMonad Set where
  constraint J Set a = Ord a
  unit x = Set.singleton x
  extend f x = Set.unions (Prelude.map f (Set.toList x))
```

4.3 Relative Comonads

Comonads are the dual structure to monads and have been used in functional programming for structuring dataflow programming and streams [21], attribute evaluation [19], array computations [13], context-dependent computation [20], and more [9].

Comonads can be described in Haskell via the following type class:

```
class Comonad c where
  coreturn :: c a → a
  (⇒⇒) :: c a → (c a → b) → c b
```

which should satisfy the following laws, for all $f :: c x \rightarrow y$, $g :: c y \rightarrow z$:

```
[C1] x ⇒⇒ coreturn = x
[C2] coreturn (x ⇒⇒ f) = f x
[C3] x ⇒⇒ (λx' → g (x' ⇒⇒ f)) = (x ⇒⇒ f) ⇒⇒ g
```

The *Comonad* class models a comonad in *coKleisli triple* form with object mapping $c : |\mathbf{Hask}| \rightarrow |\mathbf{Hask}|$, where *coreturn* is called *counit* in more mathematical definitions and $\Rightarrow\Rightarrow$ is an infix version of the *coextension* operation [20], which will be referred to here as *cobind*.

A useful intuition for comonads is of *context-dependent* computations. A value of type $c a$ models a computation of a value a which is dependent on, or parameterised by, some other values called the *context*. The *coreturn* operation thus extracts a pure, non-context-dependent value, out of the comonad, possibly by evaluating the computation at a default context or a known context (the *current context*) packaged inside the comonad. The *cobind* operation takes a context-dependent value $c a$ and a function from context-dependent values $c a$ to non-context-dependent values b , and propagates the context-dependence by applying the function $c a \rightarrow b$ at *all possible contexts* to build a context-dependent b value, of type $c b$.

An example comonad is a *pointed array*, comprising an array paired with a particular array index known as the *cursor*; *coreturn* returns the element pointed to by the cursor, and *cobind* (\Rightarrow) provides a higher-order *convolution*-like operation, applying a function to an array at every possible index, calculating a new value for each index and building a new array from these values. It is defined:

```
data Arr i a = Arr (Array i a) i
instance Ix i  $\Rightarrow$  Comonad (Arr i) where
  coreturn (Arr arr c) = arr ! c
  (Arr x c)  $\Rightarrow$  f = let es' = Prelude.map (\(i, _)  $\rightarrow$  (i, f (Arr x i))) (assocs x)
    in Arr (array (bounds x) es') c
```

The following example usage applies a discrete Laplace operator *laplace1D* over a one-dimensional array using *cobind*:

```
laplace1D (Arr a i) = if (i > 0 and i < n)
  then a ! (i - 1) - 2 * (a ! i) + a ! (i + 1)
  else 0.0

n = length inputData
x = Arr (array (0, n) inputData) 0
x' = x  $\Rightarrow$  laplace1D
```

where *inputData* :: [(Int, Float)] is provided.

Under the context-dependent understanding of comonads, *Arr i a* represents a value of type *a* that is dependent upon an index (of type *i*). For more on comonadic arrays see [13].

Above *cobind* is defined using various functions belonging to the *IArray* class, which provides an interface on array data types, with the following signatures:

```
(!)    :: (IArray a e, Ix i)  $\Rightarrow$  a i e  $\rightarrow$  i  $\rightarrow$  e
assocs :: (IArray a e, Ix i)  $\Rightarrow$  a i e  $\rightarrow$  [(i, e)]
array  :: (IArray a e, Ix i)  $\Rightarrow$  (i, i)  $\rightarrow$  [(i, e)]  $\rightarrow$  a i e
```

For the built-in *boxed* array data type *Array*, used above, there is an instance of this class which is polymorphic in the element type:

```
instance IArray Array e where ...
```

The polymorphism implies that an array may be of any element type, thus *Array* is an endofunctor $\mathbf{Hask} \rightarrow \mathbf{Hask}$, and is also a monad and a comonad.

However, the *unboxed* array type *UArray* does not have an instance of *IArray* that is polymorphic in the element type. Instead, there is a limited number of element types allowed inside of an unboxed array, with instances such as *IArray UArray Bool*, *IArray UArray Int*, *IArray UArray Float*, etc.

The unboxed array type *UArray* is thus a functor from a subcategory of element types defined by the *IArray UArray* instances. Subsequently, *UArray*

cannot form a comonad, as it is not an endofunctor. However, as with monads, *comonads need not be endofunctors*³. Relative monads can be dualised to *relative comonads* such that a comonad-like structure can be defined over a functor $K : \mathcal{K} \rightarrow \mathcal{C}$, where \mathcal{K} and \mathcal{C} are distinct categories.

Definition 5. A relative comonad *dualises* a relative monad, and is defined over categories \mathcal{K} and \mathcal{C} , comprising:

- a functor $K : \mathcal{K} \rightarrow \mathcal{C}$
- an object mapping $D : |\mathcal{K}| \rightarrow |\mathcal{C}|$
- counit operation: $\epsilon : DX \rightarrow KX$
- coextension operation: $(-)^{\dagger} : (DX \rightarrow KY) \rightarrow (DX \rightarrow DY)$

with the same comonad laws as above (modulo the presence of J in the types, and $(-)^{\dagger}$ as the prefix version of (\Rightarrow)).

In the same way as the *RMonad* class, relative comonads can be modelled in Haskell using constraint families, with the following class:

```
class RComonad d where
  constraint K d a
  counit :: K d x => d x -> x
  coextend :: (K d x, K d y) => (d x -> y) -> d x -> d y
```

Here the constraint $K d$ provides a subcategory which is the source of the relative comonad, thus $\mathcal{K} = K d$, and $\mathcal{C} = \mathbf{Hask}$. The object mapping $D : |K d| \rightarrow |\mathbf{Hask}|$ is provided by the parametric data type d , and the inclusion functor for the $K d$ subcategory of \mathbf{Hask} provides the functor $K : K d \rightarrow \mathbf{Hask}$ (although as before the inclusion functor is implicit in the type signatures).

Unboxed arrays can be defined as a relative comonad thus:

```
data UArr i a = UArr (UArray i a) i
instance Ix i => RComonad (UArr i) where
  constraint K (UArr i) a = IArray UArray a
  counit (UArr arr c) = arr ! c
  coextend f (UArr x c) =
    let es' = Prelude.map (\(i, _) -> (i, f (Arr x i))) (assocs x)
    in UArr (array (bounds x) es') c
```

The implementation of *counit* and *coextend* is the same as *coreturn* and \Rightarrow for the *Comonad* instance of boxed arrays *Arr*, modulo the change in constructor.

As another example, the mathematical notion of a *pointed set*, common in topology, comprising a set s with a distinguished element $x \in S$, can be defined using the efficient *Set* data type as a relative comonad:

³ Clarification: comonads *must* be endofunctors, but *relative comonads* can be constructed from functors with distinct source and target categories. This statement is intended to parody the title of [3].

```

data PSet a = PSet (Set a) a
instance RComonad PSet where
  constraint K PSet a = Ord a
  counit (PSet s a) = a
  coextend f (PSet s a) =
    PSet (Set.map (\a' → f (PSet (Set.delete a' s) a')) s) (f (PSet s a))

```

The implementation of *coextend* applies the function *f* to every possible combination of a set and a distinguished element, removing the distinguished element from the rest of the set using *delete*.

5 Constraint Kinds for Constraint Families

The recent *constraint kinds* extension [4] negates the need for a major syntactic and semantic extension to GHC in order to gain type-indexed constraints. The extension unifies constraints and types, where constraints become type-terms of a special kind: *Constraint*. For example, the *Eq* class constructor becomes a type constructor of kind: $* \rightarrow \text{Constraint}$. Conjunctions of constraints and empty constraints, are also now types but of the kind *Constraint*⁴ e.g.

- (*Show a, Ord a*) :: $* \rightarrow \text{Constraint}$
- () :: *Constraint*

Previously in Haskell type signatures were of the form $C \Rightarrow \tau$ where *C* denoted a constraint term and τ a type term. Now, the term appearing on the left of an arrow \Rightarrow is a type term of kind *Constraint*.

Given constraints as types, any type system features on type terms can now be reused on constraints. Thus, constraint families can be easily emulated with type families of *Constraint*-kinded types.

For example, the new *Functor* class in Section 4.1 with constraint families:

```

class Functor f where
  constraint SubCat f a = ()
  fmap :: (SubCat f a, SubCat f b) ⇒ (a → b) → f a → f b

```

can be emulated with constraint-kinded type families:

```

class Functor f where
  type SubCat f a :: Constraint
  type SubCat f a = ()
  fmap :: (SubCat f a, SubCat f b) ⇒ (a → b) → f a → f b

```

The only differences between this definition and the previous are:

⁴ Depending on the context, the tuple construction can be on types or constraints, thus $() :: *$ or $() :: \text{Constraint}$ for the unit type or empty constraint, and $(,) :: * \rightarrow * \rightarrow *$ or $(,) :: \text{Constraint} \rightarrow \text{Constraint} \rightarrow \text{Constraint}$ for the tuple type or conjunction of constraints. Currently in GHC/Haskell there is no way to describe a kind-polymorphic signature for tuple or empty tuple terms.

1. the use of the **type** keyword, instead of **constraint**, to define *SubCat* as a type family;
2. the explicit kind signature on *SubCat*, specifying that *SubCat* is a type family of constraint-kinded types.

SubCat is therefore a type family of kind $(* \rightarrow *) \rightarrow * \rightarrow \text{Constraint}$; the first parameter is the parametric data type of kind $* \rightarrow *$.

6 Related Work & Conclusion

Related Work Nogueira discussed briefly the interpretation of type classes as subcategories [12], looking at data types which are “mappable”, noting that they may be constrained in their source to a subcategory **S** and thus are functors **S** \rightarrow **ADT** (where **ADT** is used as the category of (algebraic) data types). A general definition of functors, parameterised by their subcategory, is not used.

Hughes tackled similar issues to those dealt with by constraint families, proposing that constraints on a data type be given with its definition [8] e.g.:

```
data Ord a  $\Rightarrow$  Set a = ...
```

Under Hughes’ work, the *Ord* constraint need not appear in the types of any operations e.g. *Set.map* $:: (a \rightarrow b) \rightarrow \text{Set } a \rightarrow \text{Set } b$. Hughes’ proposal is subsumed by both constraint families and constraint kinds, which provide greater flexibility for defining constraints. From the perspective of this paper, a minor benefit of Hughes’ proposal is that the data type itself provides its subcategory, as opposed to its operations/constructors enforcing a subcategory. The latter situation requires some examination of these operations to ascertain if the data type does in fact have a subcategory domain.

The RMonad library by Sittampalam and Gavin [18] provided restricted alternatives to *functor* and *monad* classes, requiring manual encoding and passing of constraints using GADTs. The *RFunctor* and *RMonad* definitions are isomorphic to our non-endofunctor and relative monad definitions, and model the same structures, although the approach here is more elegant. The mathematical understanding of these structures was not previously identified.

Bolingbroke provides a class *RMonad* for a “restricted monad” using constraint-kinded type families, which is the same as the relative monad class shown here with constraint families [4].

The original constraint families paper [14] did not provide the mathematical insights of this paper, although gave a similar definition for a class of functors with indexed constraints, and an instance for *Set*, as in Section 4.1.

Concluding Remarks One of the cornerstones of functional programming is parametric polymorphism. Many polymorphic data types are endofunctors in the category theoretical sense, and can be made an instance of the *Functor* type class in Haskell, which models endofunctors. Some of these data types are also

monads, or comonads, or both, including lists, trees, arrays, functions (with some restrictions for comonads), and products (with some restrictions for monads).

Some parametrically polymorphic data types are restricted in their polymorphism to a smaller set of types by constraints on their constructors, destructors, and transformation operations. In Haskell, type class constraints are used to restrict the polymorphism of such data structures. Usually such restrictions occur because of some *implementational* details, such as an efficient encoding of sets as balanced binary trees, requiring comparison operations on the element type, or unboxed internal representations allowing only simple, primitive types as elements (as with unboxed arrays).

Such restricted data types are still extremely useful in programming and have corresponding categorical interpretations. In the categorical interpretation of Haskell, full polymorphism corresponds to quantification over the objects of `Hask`, but restricted polymorphism (via type class constraints) corresponds to quantification over the objects of a subcategory S of `Hask`.

Until now implementation details have threatened to plague the clean, mathematical descriptions of abstract structures in programming. There is a neat mathematical model for such efficiently implemented data structures via the use of structures with subcategory domains, such as functors $S \rightarrow \mathbf{Hask}$ where $S \sqsubseteq \mathbf{Hask}$, relative monads, and relative comonads. This paper has described these and shown how the theory can be applied in Haskell, using constraint families (or constraint-kinded type families), providing the benefits of clear mathematical structuring to programming with efficient data structures.

Acknowledgments

Thank you to Alan Mycroft and Robin Message for comments on a draft of this paper. Any remaining errors or infelicities are my own. Thank you to Max Bolingbroke for his implementation of constraint kinds in GHC. This research was supported by an EPSRC Doctoral Training Award.

References

1. S. Adams. Functional Pearls: Efficient sets—a balancing act. *Journal of functional programming*, 3(4):553–562, 1993.
2. T. Altenkirch, J. Chapman, and T. Uustalu. Relative monads formalised. *To appear in the Journal of Formalized Reasoning*. *Final version pending*.
3. T. Altenkirch, J. Chapman, and T. Uustalu. Monads need not be endofunctors. *Foundations of Software Science and Computational Structures*, pages 297–311, 2010.
4. M. Bolingbroke. Constraint Kinds for GHC, 2011. <http://blog.omega-prime.co.uk/?p=127> (Retrieved 14/09/11).
5. J. Carette, O. Kiselyov, and C. Shan. Finally Tagless, Partially Evaluated. In Z. Shao, editor, *APLAS*, volume 4807 of *LNCS*, pages 222–238. Springer, 2007.

6. M. M. T. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 241–253, New York, NY, USA, 2005. ACM.
7. N. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 206–217. ACM, 2006.
8. J. Hughes. Restricted Data Types in Haskell. In *Proceedings of the 1999 Haskell Workshop. Technical Report UU-CS-1999-28*, Utrecht, 1999.
9. R. B. Kieburtz. Codata and Comonads in Haskell, 1999.
10. E. Manes. *Algebraic theories*. Springer, 1976.
11. C. McBride and R. Paterson. Functional pearl: Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.
12. P. Nogueira. When is an abstract data type a functor? *Trends in Functional Programming*, 7:217, 2007.
13. D. Orchard, M. Bolingbroke, and A. Mycroft. Ypnos: Declarative, Parallel Structured Grid Programming. In *DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, pages 15–24, NY, USA, 2010. ACM.
14. D. Orchard and T. Schrijvers. Haskell Type Constraints Unleashed. *Functional and Logic Programming*, pages 56–71, 2010.
15. S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: Type inference for generalised algebraic data types. Technical report, July 2004.
16. D. Piponi, October 2009 (Retrieved July, 2011). <http://blog.sigfpe.com/2009/10/what-category-do-haskell-types-and.html>.
17. T. Schrijvers, S. P. Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. *SIGPLAN Not.*, 43(9):51–62, 2008.
18. G. Sittampalam and P. Gavin. rmonad: Restricted monad library, 2008. <http://hackage.haskell.org/package/rmonad>.
19. T. Uustalu and V. Vene. Comonadic functional attribute evaluation. *Trends in Functional Programming-Volume 6*, pages 145–160, 2007.
20. T. Uustalu and V. Vene. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, 2008.
21. T. Uustalu and V. Vene. The Essence of Dataflow Programming. *Lecture Notes in Computer Science*, 4164:135–167, Nov 2006.
22. P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM, 1992.
23. P. Wadler. Monads for functional programming. *Advanced Functional Programming*, pages 24–52, 1995.