Categorical Programming for Data Types with Restricted Parametricity

Dominic Orchard and Alan Mycroft

Computer Laboratory, University of Cambridge
{firstname}.{lastname}@cl.cam.ac.uk

Abstract. Many concepts from category theory have proven useful tools for program abstraction, particularly in functional programming. For example, many parametric data types admit operations which are analogous to a functor and a monad. However, some parametric data types whose operations are restricted in their parametricity are not amenable to traditional category-theoretic abstractions in Haskell, despite appearing to have the right structure. This paper explains the limitations of various traditional category-theoretic approaches in Haskell, giving a precise account of the category-theoretic analogy they provide and the implications of restricted parametricity arising from ad-hoc polymorphism provided by type classes in Haskell. Following from this analysis, we generalise Haskell’s notions of functors, monads and comonads, making use of GHC’s new constraint kinds extension, providing techniques for structuring programs with both unrestricted and restricted polymorphism.

Many concepts from category theory have been adopted as design patterns for abstraction in programming; we term this approach categorical programming. For example, the notion of a functor is used to abstract map-like operations over parametric data types. In Haskell, “functors” are traditionally defined by a parametric data type together with an instance of the type class Functor:

\[
\text{class } \text{Functor } f \text{ where } \text{fmap} :: (a \to b) \to f \ a \to f \ b
\]

The prototypical example is the list data type with the standard map operation:

\[
\text{instance } \text{Functor } [] \text{ where } \text{fmap} = \text{map}
\]

The categorical laws of functors must be checked by hand if functorial behaviour is expected since Haskell has no mechanism for expressing or enforcing such laws. In general, such laws tend to hold only for a strict subset of the language; Functor defines a helpful analogy or model, rather than an actual mathematical functor.

Many parametric data types have a map-operation defining a valid Functor instance. However, data types with a map-operation restricted in its parametricity cannot be instances of Functor. For example, the Set data type in Haskell is implemented efficiently using balanced binary-trees thus many of its operations require elements of a set to be orderable such that the internal tree-representation can be balanced [1]. Set has a map-operation of type:
\( \text{Set.map} :: (\text{Ord } a, \text{Ord } b) \Rightarrow (a \rightarrow b) \rightarrow \text{Set } a \rightarrow \text{Set } b \)

where the type class constraints \((\text{Ord } a, \text{Ord } b)\) restrict the parametricity of \(\text{Set.map}\) to types with orderable values. Since \(\text{fmap}\) has no constraints on \(a\) and \(b\), the type checker rejects a \(\text{Set}\)-instance of \(\text{Functor}\) with \(\text{fmap} = \text{Set.map}\); the signature of \(\text{fmap}\) describes \textit{parametrically polymorphic} functions whilst \(\text{Set.map}\) is \textit{ad-hoc polymorphic}, with restricted polymorphism and type-dependent behaviour. Many data types similarly have a functorial \textit{map}-operation that cannot define an instance of \(\text{Functor}\) because of constraints to the parameter types.

The first contribution of this paper makes precise the categorical analogies provided by \textit{categorical programming} in Haskell using functors and related structures such as monads. Section 1 provides an overview of categorical programming and defines an interpretation for the type structure of programs, in particular for polymorphic and restricted parametricity. The interpretation elucidates the underlying mathematical mismatch between \(\text{Functor}\) and \(\text{Set.map}\): \(\text{Functor}\) is limited to \textit{endofunctors} (functors with the same source and target category) whilst \(\text{Set}\) is a functor mapping from the \(\text{Ord}\)-subcategory of Haskell. Thus ‘\(\text{Functor}\)’ is a misnomer, even as an analogy.

A full interpretation of ad-hoc polymorphism in Haskell is not given. Instead we focus on the restricted parametricity provided by ad-hoc polymorphism, ignoring the semantics of type-dependent behaviour by considering just the types of programs. Ad-hoc polymorphism over arbitrary higher-order kinded types is not considered—only the subset of categorical programming with quantification over types of kind \(*\).

Following the analysis of \(\text{Functor}\), the new \textit{constraint kinds} extension to GHC [4], coupled with Haskell’s \textit{type families}, is used to define a more general type class of functors not limited to just \textit{endofunctors} but allowing functors over \textit{subcategories}, such as \(\text{Set}\) and other such restricted data types (Section 2).

Other commonly used categorical notions in Haskell built upon the concept of functors, such as \textit{monads} and \textit{comonads}, similarly do not permit instances for data types with restricted parametric operations. The usual categorical definition of monads and comonads is in terms of endofunctors although monads can be extended to non-endofunctors, called \textit{relative monads}, as shown by Altenkirch et al. [2, 3]. Section 3 applies the techniques of Section 2, defining classes of \textit{relative monads} and their dual \textit{relative comonads} (new in this paper) with examples. The functor and relative monad examples shown here appear elsewhere under the names \textit{restricted functors} and \textit{restricted monads} [4, 22]. Our contribution makes precise these constructions from a mathematical perspective.

Section 4 concludes with some discussion of generalisations, \textit{free theorems} that can be deduced for non-endofunctors, and related work.

Familiarity with Haskell up to type classes, and the basic concepts of \textit{category}, \textit{functor}, and \textit{natural transformation}, is assumed. Readers unfamiliar with the category theory might first read an introductory text such as Fokkinga’s [7].
1 Categorical Programming in Haskell

We see two distinct approaches to applying category theory in programming: categorical semantics and categorical (or category-oriented) programming.

Categorical semantics uses category theory as a metalanguage for defining the semantics of a language, where terms are interpreted in a category that has enough additional structure to satisfy the language properties. For example, Lambek and Scott showed the semantics of well-typed terms in the simply-typed \( \lambda \)-calculus can be interpreted as morphisms in a cartesian-closed category [14].

Categorical programming uses category-theoretic concepts as design patterns and analogies for organising, structuring, and abstracting programs, simplifying both definitions and reasoning. For example, the concept of a monad is used in functional programming to abstract the composition of effect-producing expressions. Whilst categorical semantics provides a categorical interpretation to both the type and term structure of a program, categorical programming instead provides a shallow category-theoretic interpretation of just the type structure, providing a framework for structuring programs using category-theoretic concepts.

Most languages do not have a well-defined categorical semantics but still admit some form of categorical programming, albeit with some approximation to the expected axioms. However, a language with a well-defined categorical semantics will likely yield a more precise form of categorical programming, in terms of correctness and reasoning, where the categorical laws of the semantics transfer to the categorical concepts used in programs.

For Haskell, Danielsson et al. showed that a subset of monomorphic programs, without general recursion (i.e. without \( \perp \)) and without advanced features such as a type classes, has a categorical semantics in terms of bicartesian-closed categories (bcccs) providing an account of functions, products, and sums [6]. Since non-productive non-termination is usually unintended within a program, the axioms of a bccc may be assumed for the majority of programs in the full Haskell language. Coupled with its applicative syntax and powerful abstraction mechanisms, Haskell is therefore well-suited to categorical programming. However, to be clear, the categorical programming technique frequently deals in analogy rather than actual mathematics.

A categorical interpretation for categorical programming in Haskell is now defined, allowing traditional category-theoretic structures used in Haskell to be analysed in Section 1.2 and suitably generalised in Section 2.

1.1 Categorical interpretation for categorical programming

Notation: for a category \( C \), its collections of objects and morphisms are \( C_0 \) and \( C_1 \). In Haskell types, universal quantification will be explicit, \( \sigma \) and \( \tau \) range over types, \( a \) ranges over type variables (potentially subscripted) and vector notation denotes multiple syntactic elements e.g. \( \bar{a} \) is a group of type variables \( a_0 \ldots a_{|\bar{a}|} \).

The traditional approach of categorical semantics interprets types as objects of a category and well-typed terms, in a free-variable context, as morphisms from
the type of the context to the type of the expression’s result [14]. For categorical programming, we instead provide a more shallow interpretation of the type structure of programs into an imaginary category, Hask, with (monomorphic) Haskell types (of kind \(\ast\)) as objects and Haskell functions as morphisms (any non-function expression is taken as a morphism from the unit value/type (\(\mathit{()}\)). The interpretation, given by \([-]\), thus maps types to \(\mathsf{Hask}_0\) and functions to \(\mathsf{Hask}_1\) such that \(\llbracket f :: \sigma \rightarrow \tau \rrbracket : \llbracket \sigma \rrbracket \rightarrow \llbracket \tau \rrbracket \in \mathsf{Hask}_1\) where \(\llbracket \sigma \rrbracket, \llbracket \tau \rrbracket \in \mathsf{Hask}_0\).

Note that Haskell does not provide an actual category \(\mathsf{Hask}\) (with its functions as morphisms) as the axioms of a category are frequently violated in the presence of general recursion (infinite behaviour). We take \(\mathsf{Hask}\) to be our analogy for categorical programming purposes, but readers should not think that Haskell readily provides such a category in precise terms.

In \(\mathsf{Hask}\), every pair of objects (i.e., types) \(a, b\) has an object of the type of functions from \(a\) to \(b\) called the hom-object, denoted \(\mathsf{Hask}((a, b))\). A category with hom-objects for every pair of objects is called closed. Such categories allow higher-order functions to be interpreted as morphisms from hom-objects.

For constant, tuple, and function types, \([-]\) is defined recursively:

\[
\begin{align*}
\llbracket c \rrbracket &= c : \mathsf{Hask}_0 \\
\llbracket \sigma \rightarrow \tau \rrbracket &= \mathsf{Hask}(\llbracket \sigma \rrbracket, \llbracket \tau \rrbracket) : \mathsf{Hask}_0 \iff \llbracket \sigma \rrbracket : \mathsf{Hask}_0 \land \llbracket \tau \rrbracket : \mathsf{Hask}_0 \\
\llbracket (\sigma, \tau) \rrbracket &= \llbracket \sigma \rrbracket \times \llbracket \tau \rrbracket : \mathsf{Hask}_0 \iff \llbracket \sigma \rrbracket : \mathsf{Hask}_0 \land \llbracket \tau \rrbracket : \mathsf{Hask}_0
\end{align*}
\]

**Parametric Polymorphism** Many category-theoretic concepts are defined universally over the objects of a category i.e. “for every object \(X\) in \(C\) [...]”. Since the objects of \(\mathsf{Hask}\) are Haskell types, parametric polymorphism is central to categorical programming, providing universal quantification over types. Accordingly, only polymorphism over types of kind \(\ast\) will be considered here.

The polymorphic \(\lambda\)-calculus can be given a categorical semantics in terms of indexed categories, giving a semantics to type abstraction and application [21]. For our purposes, expanding upon an entire categorical semantics for polymorphism in Haskell is not necessary. Instead, polymorphic types will be interpreted by indexed families. An \(X\)-indexed family of \(Y\) (written \(X \mapsto Y\)) has an \(X\)-element associated to each \(Y\)-element in the image. A family \(f : X \mapsto Y\) will be defined as \(f_x = y\), where \(x\) is a variable of an \(X\)-element and \(y\) is a \(Y\)-element.

The interpretation of polymorphic types is provided at the top-level by \([-]^{\forall}\) and within the binding scope of the universal quantifier by \([-]^{\forall}_{\Gamma}\) parameterised by a sequence \(\Gamma\) of indices for each free variable of the type it interprets (with some arbitrary canonical order). The definition of \([-]^{\forall}_{\Gamma}\) is the same as \([-]\) in (1), with \(\Gamma\) passed to recursive sub-terms, with an additional rule for type variables:

\[
\begin{align*}
\llbracket [a]^{\forall}_{\Gamma} &= a : \mathsf{Hask}_0 \\
\llbracket [\forall \sigma. \tau]^{\forall}_{\Gamma} &= (\prod_{\tau} \mathsf{Hask}_0) \mapsto \mathsf{Hask}_0 \iff \llbracket \tau \rrbracket : \mathsf{Hask}_0
\end{align*}
\]

---

1 Indexed families here differ to the notion of type families in GHC/Haskell, which are *partial* and where the types in the image are potentially unrelated by any common structure; they are not *parametric* in the sense of Reynolds [20].
where \( \prod Hask \) is the product category \((Hask \times \ldots \times Hask)\) for the \( |\pi|\)-times product of \( Hask \). The full definition of a product category is irrelevant here since only objects are used. Most importantly, product categories have the property that \((C \times D)_0 \equiv (C_0 \times D_0)\) thus a multi-variable polymorphic type is interpreted as family indexed by a product (tuple) of types.

If \( \pi = \emptyset \) then \( \prod_0 Hask = 1 \), where \( 1 \) is the unit category with a single object and \( 1_0 \mapsto Hask_0 \equiv Hask_0 \). Thus for a type with no type variables the polymorphic interpretation collapses to the monomorphic.

As an example, the polymorphic type of the \( \text{fst} \) function is interpreted:

\[
\forall a . b . a \to (a, b)_{x,w} = Hask(x, (x, y)) : (Hask \times Hask)_0 \mapsto Hask_0
\]

Polymorphic functions are interpreted as indexed families of morphisms:

\[
\left[ f :: \forall x . \sigma \to \tau \right]_{x} = \left[ \forall x . \sigma \right]_{\sigma} \to \left[ \forall x . \tau \right]_{\tau} : (\prod_{|\pi|} Hask)_0 \mapsto Hask_1 \tag{3}
\]

**Parametric type constructors** We consider only parametric types with a single parameter, e.g. \( \text{data} \ F a = \ldots \) defining a type constructor \( F \) of kind \( \star \to \star \) i.e. \( F \) maps a type, of kind \( \star \), to another type. Parametric polymorphic type constructors will be interpreted similarly to universally quantified types:

\[
[F]_r = F : Hask_0 \mapsto Hask_0 \text{ iff } \text{data} \ F a = \ldots \ \ \ \ \ \ \ [\sigma \tau]_r = [\sigma]_r [\tau]_r : Hask_0 \mapsto Hask_0 \wedge [\tau]_r : Hask_0 \tag{4}
\]

We will treat data types as abstract since we are not concerned with the properties of particular data types, only the type constructors.

**Type classes and ad-hoc polymorphism** Rather than give a precise categorical semantics for type classes in general we distinguish two specific uses of type classes relevant to this paper: single parameter type classes parameterised by 1). nullary types of kind \( \star \) and 2). type constructors of kind \( \star \to \star \). Classes parameterised by nullary types are pertinent as they provide type class constraints over \( Hask \) objects. Classes parameterised by type constructors are used to abstractly define concepts such as functors, monads, etc. in Haskell.

1). **Classes parameterised by nullary types** The instances of a type class, \( \text{class} \ S a \) (where \( a \) has kind \( \star \)), define a subset of \( Hask \) objects: the types which have an instance of \( S \). A polymorphic type variable with a type class constraint, e.g., \( \forall a . S a \Rightarrow \tau \), is therefore restricted in its quantification. Such types will be interpreted as an indexed family \( S_0 \mapsto Hask_0 \) where \( S \) is the subcategory of \( Hask \) whose objects are only those with an instance of \( S \) and whose morphisms are between types with instances of \( S \). The corresponding subcategory of a type class will be written in sans font. Formally subcategories are defined:

**Definition 1.** For a category \( C \), a subcategory \( S \) of \( C \) comprises a subclass of the objects of \( C \) and a subclass of the morphisms of \( C \) such that:

- for every morphism \( f : X \to Y \in S_1 \) then \( X, Y \in S_0 \);
- for every morphism pair \( f : X \to Y, g : Y \to Z \in S_1 \) then \( g \circ f : X \to Z \in S_1 \);
- for every object \( X \in S_0 \) there is an identity morphism \( \text{id}_X : X \to X \in S_1 \).

The parameters of a type class can be constrained, with so-called superclass constraints, implying a partial ordering relation \( \sqsubseteq \) between the corresponding subcategories, for which \text{Hask} is the upper bound. For example, the definition of \text{Ord} has a superclass constraint: class \text{Eq} \ a \Rightarrow \text{Ord} \ a \ where ..., thus any type that is an instance of \text{Ord} must be an instance of \text{Eq}, i.e., \text{Eq} is the superclass of \text{Ord} thus \text{Ord} \sqsubseteq \text{Eq} \sqsubseteq \text{Hask}. Every subcategory \( S \) of \text{C} has an inclusion functor \( I : S \to C \) mapping objects and morphisms into its supercategory.

A type with multiple constraints e.g. \( \forall a. (S_a, T_a) \Rightarrow \tau \) will be interpreted as an \( (S \sqcap T)_0 \mapsto \text{Hask}_0 \) family, where \( S \sqcap T \) is the intersection category which has only the objects and morphisms that are in both \( S \) and \( T \).

Using subcategories to interpret restricted (ad-hoc) polymorphism, the full interpretation of restricted polymorphic types is provided at the top-level by \([[-]]^{\forall \Rightarrow} \) and within the binding of the universal quantifier and type constraints by \([[-]]^{\Delta_\tau} \) where \( \Delta_\tau \) is the set of type classes for which there are type class constraints over \( \tau \) in the interpreted type. The definition of \([[-]]^{\Delta_\tau} \) is given in Figure 1 where \((\bigcap_{\sigma \in \Delta_\tau} S)\) is the intersection subcategory \( S^0 \sqcap \ldots \sqcap S^n \) for each class \( S^i \in \Delta_\tau \), i.e., the intersection category of all subcategories corresponding to class constraints on \( \tau \). If \( \Delta_\tau = \emptyset \) i.e. \( \tau \) is unconstrained, then \( \bigcap_{S \in \Delta} S = \text{Hask} \). Thus for a type with no constraints, the ad-hoc polymorphic interpretation collapses to the polymorphic interpretation.

\[
[a]^{\Delta_\tau} = a : (\bigcap_{\sigma \in \Delta_\tau} S)_0 \quad \text{iff} \quad a \in \Gamma \tag{5}
\]

\[
[e]^{\Delta_\tau} = e : (\bigcap_{\sigma \in \Delta_\tau} S)_0 \tag{6}
\]

\[
[[\sigma, \tau]]^{\Delta_\tau} = [\sigma] \times [\tau] : (\bigcap_{\sigma \in \Delta_{(\sigma, \tau)}} S)_0 \quad \text{iff} \quad [\sigma]^{\Delta_\tau} : C_0 \land [\tau]^{\Delta_\tau} : D_0 \tag{7}
\]

\[
[[\sigma \to \tau]]^{\Delta_\tau} = (C \sqcap D) : ([\sigma]^{\Delta_\tau}, [\tau]^{\Delta_\tau}) : (\bigcap_{\sigma \in \Delta_{(\sigma \to \tau)}} S)_0 \quad \text{iff} \quad [\sigma]^{\Delta_\tau} : C_0 \land [\tau]^{\Delta_\tau} : D_0 \tag{8}
\]

\[
[[\sigma \tau]]^{\Delta_\tau} = [[\sigma]]^{\Delta_\tau} \times [[\tau]]^{\Delta_\tau} : (\bigcap_{\sigma \in \Delta_{(\sigma \tau)}} S)_0 \quad \text{iff} \quad [\sigma]^{\Delta_\tau} : \text{Hask}_0 \mapsto \text{Hask}_0 \land [\tau]^{\Delta_\tau} : C_0 \tag{9}
\]

**Fig. 1.** Categorical interpretation of ad-hoc polymorphic types in Haskell.

Interpretations of type variables (5), constant types (6), and tuple types (7) resemble the monomorphic and polymorphic interpretations, but now each type is an object of a subcategory corresponding to its constraints. Constraints over type constructors are not interpreted, since we consider only constraints over types of kind \(*\), thus the interpretation of type constructors is as before (4). Type constructor application (9) is interpreted as an object of a subcategory, but the type constructor is still a \text{Hask}_0 \mapsto \text{Hask}_0 \) family.

For tuples (7) and constructor applications (9), the subcategories of the sub-types \( \sigma \) and \( \tau \) do not affect the subcategory of the constructed type since the constructors apply to any objects in \text{Hask} thus any objects of any \text{Hask}-subcategory.
For function types with source and target types in the same subcategory, the hom-object interpretation can be refined from $\text{Hask}$ (as in (1)) to the subcategory in which both objects reside. The interpretation of function types (8) generalises further, where a function type with source in $C$ and target in $D$ is a hom-object of $C \sqcap D$, the least upper bound category as defined by $\sqsubseteq$ where $C \sqcap C = C$. Note that a hom-object of a subcategory is not necessarily in the subcategory itself.

Restricted (ad-hoc) polymorphic functions are interpreted similarly to polymorphic functions (3) as indexed families of morphisms:

$$\llbracket f \rrbracket \vDash \forall \alpha. S^\tau \Rightarrow \sigma : C_0 \rightarrow \llbracket \forall \alpha. S^\tau \Rightarrow \tau \rrbracket_D : (\prod_{\alpha \in \pi} \bigcap_{\sigma \in \Delta \alpha} S)(0) \mapsto (C \sqcap D)_1$$  (10)

2). Type classes parameterised by type constructors Type classes with a single parameter, e.g. $\text{class} F f$, where $f : * \rightarrow *$ will be treated, more informally than the other concepts in this section, as meta-mathematical definitions of an abstract $F$-structure comprising an indexed family $f : \text{Hask}_0 \mapsto \text{Hask}_0$ together with some indexed families of morphisms (the class methods) related to the parameter indexed family.\(^2\) An instance of the class provides an instance of the structure by providing an indexed family and instances of the operations.

1.2 Interpretation of $\text{Functor}$ and restricted $\text{map}$-operations

$\text{Functor}$ Since $\text{Functor}$ is parameterised by a type constructor $f$ of kind $* \rightarrow *$ it is interpreted as a meta definition, thus $\text{class} \ F \ f$ where $f : * \rightarrow *$ will be treated, more informally than the other concepts in this section, as meta-mathematical definitions of an abstract $F$-structure comprising an indexed family $f : \text{Hask}_0 \mapsto \text{Hask}_0$ together with some indexed families of morphisms (the class methods) related to the parameter indexed family.\(^2\) An instance of the class provides an instance of the structure by providing an indexed family and instances of the operations.

$$\llbracket f \rrbracket \vDash \forall \alpha. S^\tau \Rightarrow \sigma : C_0 \rightarrow \llbracket \forall \alpha. S^\tau \Rightarrow \tau \rrbracket_D : (\prod_{\alpha \in \pi} \bigcap_{\sigma \in \Delta \alpha} S)(0) \mapsto (C \sqcap D)_1$$  (11)

In comparison, a category-theoretic functor $F : C \rightarrow D$ is defined by:

- an object mapping $F_0 : C_0 \rightarrow D_0$ mapping all objects $A \in C_0$ to $F_0 A \in D_0$;
- and a morphism mapping $F_1 : C_1 \rightarrow D_1$ mapping all morphisms $f : X \rightarrow Y \in C_1$ to $F_1 f : F_0 X \rightarrow F_0 Y \in D_1$

with the usual functorial axioms. An object mapping is equivalent to an object-indexed family of objects, thus in the interpretation of $\text{Functor}$, the class parameter $f$ matches the object mapping in the categorical definition where $C = D = \text{Hask}$. The interpretation of $\text{ fmap}$ is a morphism in $\text{Hask}$ over $\text{Hask}$ hom-objects, analogous to the morphism mapping in the categorical definition but embedded within $\text{Hask}$. The embedding of a functor into a category is captured by the notion of enriched, or strong [13], functors [11]. The enriched explanation is not

\(^2\) In Haskell, the type of each class method must use the class parameter so that a use of a class method can be statically resolved to a particular class instance.
discussed here for space reasons; details can be found in the author’s upcoming thesis. In brief, a strong (equivalently enriched) endofunctor $F$ on a closed (equivalently self-enriched) category $C$ has an object mapping $F_0 : C_0 \to C_0$ and an indexed family of morphisms:

$$F_{x,y} = C(x, y) \to C(F_0 x, F_0 y) : (C_0 \times C_0) \mapsto C_1$$

corresponding exactly to the interpretation of $\text{fmap}$ here, where $C = \text{Hask}$.

Thus, $\text{Functor}$ models strong endofunctors on $\text{Hask}$.

**Restricted map-operations** The introduction gave the type of the $\text{Set.map}$ function which is constrained in the parameter types to the $\text{Ord}$ class. The general form of the signature is, for some data type $F$ and class $S$:

$$F\.\text{map} :: (S a, S b) \Rightarrow (a \to b) \to F a \to F b$$

which is interpreted as $[F]^{\forall a,b} : \text{Hask}_0 \mapsto \text{Hask}_0$ and:

$$[F\.\text{map}]^{\forall a,b} = S(a,b) \to T(fa,fb) : (S \times S)_0 \mapsto \text{Hask}_1$$

Thus $F\.\text{map}$ maps from morphisms of the subcategory $S$. Although the type constructor $F$ is the family $\text{Hask}_0 \mapsto \text{Hask}_0$, the interpretation of $F\.\text{map}$ implies that $F$ is essentially a strong functor $F : S \to \text{Hask}$, i.e., not an endofunctor on $\text{Hask}$ as captured by $\text{Functor}$. Thus $F\.\text{map}$ cannot be used to define an $F$-instance of $\text{Functor}$ due to the mismatch between (11) and (12).

The next section generalises $\text{Functor}$ to non-endofunctors using recent features added to GHC/Haslkek. The (non-standard) term $\text{exofunctor}$ is used for emphasis to describe functors that need not be endofunctors.

## 2 Generalising $\text{Functor}$ from endofunctors to exofunctors

A general class of exofunctors for Haskell requires an operation which maps between arbitrary subcategories of $\text{Hask}$ in both the source (as in the $\text{Set.map}$) and in the target, i.e. an operation with the following interpretation:

$$[\text{exfmap}]^{\forall a,b} = S(a, b) \to T(fa, fb) : (S \times S)_0 \mapsto \text{Hask}_1$$

A general class of such structures requires parameters for the source and target subcategories associated with the type constructor $f$. These subcategory parameters will be expressed using type-indexed constraints in Haskell.

### 2.1 Type-indexed constraints and $\text{RFunctor}$

Type classes in Haskell fix the types of their methods with type signatures in the class declaration. The type families extension to GHC allows types of a class method to vary per-instance of a class by defining a family of types associated
with the class, indexed by its parameter, and using the family in method signatures [5]. The analogous concept of a constraint family has been previously proposed, allowing the constraints of a class method to vary per-instance by defining a family of constraints associated with a class, indexed by the class parameter [18]. Constraint families provide a solution to the Set-Functor problem.

A recent extension to GHC subsumes the constraint family proposal by re-defining constraints as types with a distinct constraint kind, thus type families may return types of kind Constraint. The constraint kinds extension,⁴ implemented by Bolingbroke [4], negates the need for a syntactic and semantic extension to the type checker to add constraint families. Under the extension, a class constructor, e.g. Ord, is a type constructor of kind * → Constraint. Depending on the context, tuples can be types or constraints i.e. (, ) :: * → * → * or (, ) :: Constraint → Constraint → Constraint (conjunction of constraints) and () :: * or () :: Constraint for the unit type or empty (true) constraint.

The Functor class can therefore be generalised using an (associated) type family of constraint-kinded types:⁴

```haskell
class RFunctor f where
type SubCats f a :: Constraint
type SubCats f a = ()
rfmap :: (SubCats f a, SubCats f b) ⇒ (a → b) → f a → f b
```

which includes a default empty constraint. Instances for Set and lists are:

```haskell
instance RFunctor Set where instance RFunctor [] where
type SubCats Set a = Ord a type SubCats [] a = ()
rfmap = Set.map rfmapi = map
```

The interpretation of RFunctor depends on the constraints specified by SubCats which vary per-instance of the class and permit constraints over a and b as well as fa and fb, for both the source and target functions. For example:

```haskell
instance RFunctor Foo where
type SubCats Foo a = (S a, T (Foo a))
```

defines subcategories for both the source and target providing an interpretation to rfmapi as described by (13). For Set, the interpretation of rfmapi is:

\[ rfmapi_{a,b}^{\text{Set}} = \text{Ord}(a, b) \rightarrow \text{Hask}(\text{Set} a, \text{Set} b) : (\text{Ord} \times \text{Ord})_{a,b} \rightarrow \text{Hask}_{1} \quad (14) \]

Another example exofunctor is the UArray type of unboxed arrays which constrains its elements to primitive types (Int, Float etc.) for which there is an efficient, unboxed storage representation. UArray has a map operation:

---

³ Constraint kinds are enabled by the pragma {-# LANGUAGE ConstraintKinds #-}. At the time of writing it is also necessary to import GHC.Prim.

⁴ This definition is the constraint-kinds analogue of the Set-Functor solution shown by Orchard and Schrijvers using constraint families [18].
amap :: (IArray UArray e', IArray UArray e, Ix i) ⇒ (e' → e) → UArray i e' → UArray i e

Thus, UArray i (for some index type i) is an exofunctor (IArray UArray) → Hask with the following instance of RFunctor:

```haskell
instance Ix i ⇒ RFunctor (UArray i) where
type SubCats (UArray i) a = IArray UArray a
rfmap = amap
```

The category-theoretic structures of monads and comonads are defined over endofunctors together with some operations (natural transformations). The next section generalises monads and comonads to exofunctors.

### 3 Relative Monads and Comonads

**Relative Monads** Monads in Haskell are traditionally defined by the class:

```haskell
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```
satisfying various laws [26]. The Monad class models a monad in Kleisli triple form defined over an object mapping $m : C_0 → C_0$ [15]. An equivalent presentation defines monads in terms of an endofunctor which can be derived from the Kleisli triple form with the following construction of the morphism mapping:

```haskell
instance Monad m ⇒ Functor m where
  fmap f x = x >>= (return ∘ f)
```

Since monads are endofunctors, data types that are not endofunctors are not monads. However, data types that are exofunctors may be relative monads, a generalisation of monads for functors $J : J → C$ where $J$ and $C$ may be distinct [3].

**Definition 2.** A relative monad over categories $J$ and $C$ comprises:

- a functor $J : J → C$
- an object mapping $T : J_0 → C_0$
- a natural transformation (unit) $η_X : JX → TX$ (analogous to return)
- a natural transformation (extend) $(−)_X^Y : (JX → TY) → (TX → TY)$ (analogous to $>>=$ but in prefix form)

with the usual monad laws (modulo the presence of $J$ in the types) [2, 3].

Relative monads can be defined in Haskell similarly to exofunctors:

```haskell
class RMonad t where
  type RSubCats t x :: Constraint
```
unit :: (RSubCats t x) ⇒ x → t x
extend :: (RSubCats t x, RSubCats t y) ⇒ (x → t y) → t x → t y

As with RFunctor, the exact interpretation of RMonad depends on the constraints specified by RSubCats which define both the subcategories J and C. The indexed family \( t : \text{Hask}_0 \mapsto \text{Hask}_0 \) of RMonad corresponds to the object mapping \( T \) of relative monads, and is constrained to \( t : J_0 \mapsto C_0 \) in the types of unit and extend by RSubCats. The two indexed families of morphisms, unit and extend, correspond to the natural transformations in the categorical definition, since natural transformations can be understood as indexed families of morphisms. For RMonad, the \( J \) functor of a relative monad is the inclusion functor \( I : J \rightarrow C \), thus \( J \subseteq C \). The inclusion functor \( I \) is implicit and elided in the type signatures for RMonad since an inclusion functor between a subcategory and supercategory does not affect the types. The property \( J \subseteq C \) affects the interpretation of an RMonad instance in two ways depending on the relationship between \( J \) and \( C \).

Consider an instance of RMonad:

```hs
instance RMonad Foo where
type RSubCats Foo x = (J x, C (Foo x))
```

There are two interpretations for unit and extend, depending on \( J \) and \( C \):

- if \( J \subseteq C \) therefore \( J \cap C = C \):

  \[
  \begin{align*}
  \text{unit}_a &= a \rightarrow \text{Foo } a :: J_0 \mapsto C_1 \\
  \text{extend}_{a,b} &= C(a, \text{Foo } b) \rightarrow C(\text{Foo } a, \text{Foo } b) : (J \times J)_0 \mapsto \text{Hask}_1
  \end{align*}
  \]

- if \( J \not\subseteq C \) therefore \( J \cap C = \text{Hask} \):

  \[
  \begin{align*}
  \text{unit}_a &= a \rightarrow \text{Foo } a :: J_0 \mapsto \text{Hask}_1 \\
  \text{extend}_{a,b} &= \text{Hask}(a, \text{Foo } b) \rightarrow C(\text{Foo } a, \text{Foo } b) : (J \times J)_0 \mapsto \text{Hask}_1
  \end{align*}
  \]

In the first case, the relative monad is over a functor \( J \rightarrow C \) and in the second over a functor \( J \rightarrow \text{Hask} \). In terms of programming, this subtlety in the interpretation does not affect the expressivity of the pattern nor introduce added complexity.

As an example instance, Set is a relative monad where the object mapping \( T \) is given by Set and RSubCats specifies that \( J = \text{Ord} \) and \( C = \text{Hask} \):

```hs
instance RMonad Set where
type RSubCats Set x = Set.singleton x
unit x = Set.singleton x
extend f x = Set.unions (Prelude.map f (Set.toList x))
```

Similarly to the definition of a functor from a monad, an exofunctor can be constructed from a relative monad:

```hs
instance RMonad m ⇒ RFunctor m where
type SubCats m a = RSubCats m a
rfmap f = extend (unit ∘ f)
```
**Relative comonads**

Comonads are the dual structure to monads and have been used in functional programming for structuring dataflow programming and streams [24], array computations [17], context-dependent computation [23], and more [12]. However, comonads are less well-known in programming than monads.

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

```haskell
class Comonad c where
  coreturn :: c a -> a
  (≫) :: c a -> (c a -> b) -> c b
```

with dual laws to those of monads [23, 24], where \(\Rightarrow\) is pronounced **cobind**.

Since comonads are less widely-used than monads we provide some intuition and a more involved example. A useful intuition for comonads is of the type \(c a\) representing a context-dependent computation, where **coreturn** evaluates the computation at a default or known “current” context and **cobind** takes a function \(c a \rightarrow b\) of an operation on a local context and applies it globally, at all contexts.

For example, the pointed array comonad comprises an array paired with a particular array index known as the **cursor** which denotes the current context of execution [17]; **coreturn** returns the array element pointed to by the cursor, and **cobind** provides a higher-order **convolution**-like operation, applying a function to an array at every possible index, calculating a new value for each index:

```haskell
data Arr i a = Arr (Array i a)
instance Ix i => Comonad (Arr i) where
  coreturn (Arr arr c) = arr ! c
  (Arr x c) ≫ f = let es' = map (\i -> (i, f (Arr x i))) (indices x)
    in Arr (array (bounds x) es') c
```

For example, the following defines a discrete Laplace operator which is applied over a one-dimensional array using **cobind** (where \(\text{inputData} :: [(\text{Int}, \text{Float})]\)):

```haskell
laplace1D (Arr a i) = if (i > 0 ∧ i < (n - 1))
  then a ! (i - 1) - 2 * (a ! i) + a ! (i + 1) else 0
n = length inputData
x = Arr (array (0, n) inputData) 0
x' = x ≫ laplace1D
```

The definition of **cobind** uses various methods of the **IArray** class, which provides an interface on array data types, for example:

```haskell
(!) :: (IArray a e, Ix i) ⇒ a i e → i → e
```

For the boxed array data type **Array** used above, there is an instance of **IArray** which is polymorphic in the element type (**instance IArray Array e**). Thus **Array** is an endofunctor \(\text{Hask} \rightarrow \text{Hask}\) and is also a monad and a comonad.

However, the unboxed array type **UArray** seen earlier does not have an instance of **IArray** polymorphic in the element type, but has a limited number of
monomorphic instances for primitive types. \texttt{UArray} is thus restricted and is an exofunctor \((\texttt{IArray} \texttt{UArray}) \to \texttt{Hask})\), therefore cannot be a comonad. However, as with monads, comonads can be generalised to \textit{relative comonads} on exofunctors.

\textbf{Definition 3.} A relative comonad dualises a relative monad, and is defined over categories \(K\) and \(C\), comprising:

- a functor \(K : K \to C\)
- an object mapping \(D : K_0 \to C_0\)
- a natural transformation (counit): \(\epsilon_X : DX \to KX\)
- a natural transformation (coextend): \((-)_{X,Y}^\dagger : (DX \to KY) \to (DX \to DY)\)

with the usual comonad laws (modulo the presence of \(K\) in the types).

Relative comonads can be defined in Haskell similarly to relative monads:

```haskell
class RComonad d where
type RCSubCats d x :: Constraint
counit :: RCSubCats d x \Rightarrow d x \to x
coextend :: (RCSubCats d x, RCSubCats d y) \Rightarrow (d x \to y) \to d x \to d y
```

As with relative monads, the functor \(K\) in the categorical definition is taken as the inclusion functor \(K \to C\) in \(RComonad\) and is implicit in the type signatures as before. The previous analysis for \(RMonad\) dualises for \(RComonad\).

Unboxed arrays can be defined as a relative comonad thus:

```haskell
data UArr i a = UArr (UArray i a)
instance Ix i \Rightarrow RComonad (UArr i) where
type RCSubCats (UArr i) x = IArray UArray x
counit (UArr arr c) = ... -- same as \texttt{coreturn} for the \texttt{Arr} comonad
coextend f (UArr x c) = ... -- same as \((\Rightarrow)\) for the \texttt{Arr} comonad
```

As another example, the notion of a \textit{pointed set} common in topology, comprising a set \(S\) with a distinguished element \(x \in S\), can be defined as a relative comonad on the efficient \texttt{Set} data type:

```haskell
data PSet a = PSet (Set a) a
instance RComonad PSet where
  type RCSubCats PSet x = Ord x
  counit (PSet s a) = a
  coextend f (PSet s a) =
    PSet (Set.map (\lambda a' \to f (PSet (Set.delete a' s) a')) s) (f (PSet s a))
```

where \texttt{coextend} applies its function to every possible combination of a set and a distinguished element, with the distinguished element removed from the set.
4 Discussion

Generalisations The constraints $RSubCats$ and $RCSSubCats$ for relative monads and comonads permit subcategory definitions of a particular style. However, $RMonad$ and $RComonad$ can be generalised to allow more interesting subcategories. For example, $RComonad$ can be generalised to:

```haskell
class RComonad d where
type RObjs d x :: Constraint
type RMorphs d x y :: Constraint
counit :: RObjs d x ⇒ d x → x
coextend :: (RMorphs d x y, RObjs d x, RObjs d y) ⇒ (d x → y) → d x → d y
```

where the specification of categories $K$ and $C$ is split between families $RObj$s and $RMorph$s. Since $counit$ involves just objects of $K$ and $C$ the single parameter $RObj$s is used, specifying objects of the subcategories. However, for $coextend$, involving objects and morphisms of $J$ and $C$, $RMorph$s offers greater flexibility for specifying the morphisms of subcategories, allowing constraints relating the source and target types of morphisms, since it is parameterised by both types. For example, a comonad $D$ can be defined on the subcategory of endomorphisms (with the same source and target object) of $Hask$ using an equality constraint [5]:

```haskell
type RMorphs D x y = x~y
```

An example comonad on the subcategory of endomorphisms is an array with regions of interior and exterior elements, where $coextend$ applies an operation over just the interior elements of the array. Such array operations need not perform bounds-checking because any out-of-bound values are provided by the exterior elements. Since the exterior elements are not transformed by $coextend$, the comonad can only be defined on the subcategory of endomorphisms so that type safety is preserved, where all elements in the array are of the same type.

Functors can be generalised to allow constraints relating the source and target types of a morphism, similarly to $RMorphs$, yielding a generalisation of functors to semi-categories (categories without identities).

Naturality and Free Theorems Building on Reynold’s analysis of parametric polymorphism using relations as types [20], Wadler showed that, in pure functional languages, theorems can be deduced about parametrically polymorphic operations from their types alone; thus “theorems for free” [25]. One such theorem is, for a polymorphic function of type $r :: [a] → [a]$ and any $f :: x → y$:

$$r \circ (\text{map } f) = (\text{map } f) \circ r$$

The theorem generalises to operations on data types which are functors, so that given $r :: F a → G a$ where Functor $F$ and Functor $G$, for any $f :: x → y$:

$$r \circ (\text{fmap } f) = (\text{fmap } f) \circ r$$
The occurrence of \textit{fmap} on the left is for the data type \textit{G} and that on the right for the data type \textit{F} and \textit{r} is instantiated at different types on the left- and right-hand sides. This theorem corresponds to the \textit{naturality} condition of a natural transformation mapping between two functors.

Wadler’s technique includes free theorems for parametric operations with their polymorphism restricted by class constraints [25, §3.4]. For example, \textit{sort} :: \textit{Ord \ a} ⇒ \textit{[\ a]} → \textit{[\ a]} has the free theorem, for any \textit{f} :: \textit{a} → \textit{b}:

\[(f \text{\textit{map} } f) \circ \text{\textit{sort}} = \text{\textit{sort}} \circ (f \text{\textit{map} } f) \text{ iff } \forall x, y :: a. x \leq y \Rightarrow (f x) \leq (f y)\]

The side condition states that \textit{f} is \textit{monotonic} with respect to the \textit{Ord \ a} and \textit{Ord b} orderings. De Bruijn included such constraints in his account of naturality from polymorphism, referring to this property as \textit{restricted naturalness} [19].

The same approach can be applied for exofunctors taking into account the constraints specifying subcategories. The free theorem for the \textit{Set} exofunctor with a natural operation has the monotonicity side condition:

Given an operation \textit{r} :: \textit{Set \ a} → \textit{Set \ a} then \forall \textit{f} :: (\textit{Ord \ a}, \textit{Ord \ b}) ⇒ \textit{a} → \textit{b} \text{ \textit{r} \circ (f \text{\textit{map} } f) = (f \text{\textit{map} } f) \circ \text{\textit{r}} \text{ iff } \forall x, y :: \textit{a}. x \leq y \Rightarrow (f x) \leq (f y)}

For example, the operation \textit{deleteMin} :: \textit{Set \ a} → \textit{Set \ a} is only natural with respect to monotonic functions since it does not compare elements itself, but simply uses the internal binary tree structure of \textit{Set} to remove the least element (the left-most element). Thus a non-monotonic function prevents the naturality property holding, as it changes the structure of the tree.

For operations \textit{r} :: (\textit{Functor \ f}) ⇒ \textit{Set \ a} → \textit{f \ a} and \textit{r} :: (\textit{Functor \ f}) ⇒ \textit{f \ a} → \textit{Set \ a} the same free theorem as above holds, plus any additional restrictions to naturality provided by the functor \textit{f}.

\textit{Related Work} Type classes have often been explained as describing sets of types (for example [10]). Nogueira discussed briefly the interpretation of type classes as subcategories [16], looking at data types which are “mappable”, noting that they may be constrained in their source to a subcategory \textit{S} and thus are functors \textit{S} → \textit{ADT} (where \textit{ADT} denotes some category of (algebraic) data types). A general definition of functors over subcategory was not provided.

Hughes tackled similar issues to those dealt with by constraint families, proposing that constraints on a data type be given along with its definition [9], e.g., \textbf{data} \textit{Ord \ a} ⇒ \textit{Set \ a} = …. Consequently, the \textit{Ord} constraint need not appear in the types of operations, e.g., \textit{Set.map} :: (\textit{a} → \textit{b}) → \textit{Set \ a} → \textit{Set \ b}. Hughes’ proposal is subsumed by constraint families which provide greater flexibility for specifying constraints. A categorical interpretation of Hughes’ approach would likely show his approach equivalent to that using constraint families.

The RMonad library by Sittampalam and Gavin [22] provided restricted alternatives to \textit{functor} and \textit{monad} classes, requiring manual encoding and passing of constraints using GADTs. The names \textit{RFunctor} and \textit{RMonad} are used in their library, which have the same structure as our definitions although the use of constraint families here is much more succinct. The mathematical understanding of these structures was not previously identified.
Our interpretation of the type structure for polymorphic and ad-hoc polymorphic types was inspired by the work of Seely using indexed categories for the second-order $\lambda$-calculus [21], and the use of indexed families in other polymorphic semantics, such as in the work of Gunter [8].

**Concluding remarks** The categorical programming approach, as a form of design pattern, is increasingly popular both inside and outside of functional programming. However, the approach is largely informal, usually proceeding by vague mathematical analogy, with little analysis. This paper elucidated the categorical programming approach more formally, linking programming language features with categorical concepts, thus providing a framework explaining the power and limitations of Haskell functors (and related structures) as design patterns. This account led to generalisations of Haskell functors, monads, and comonads, allowing categorical programming with data types whose operations are restricted in their parametricity by ad-hoc polymorphic definitions.

**Acknowledgments** Thanks to Robin Message, Tomas Petricek, and anonymous reviewers for comments on an earlier version of this paper. Thanks to Max Bolingbroke for his implementation of constraint kinds in GHC. This work was supported by an EPSRC Doctoral Training Award.

**References**