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*:

class *Functor f* **where** $\mathit{fmap} :: (a \to b) \to f\ a \to f\ b$

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

instance *Functor* [ ] **where** $\mathit{fmap} = \mathit{map}$

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:

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

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

The first contribution of this paper makes precise the categorical analogies provided by *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 *Functor* and $Set.map$: *Functor* is limited to *endofunctors* (functors with the same source and target category) whilst *Set* is a functor mapping from the *Ord-subcategory* of Haskell. Thus '*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 *Functor*, the new *constraint kinds* extension to GHC [4], coupled with Haskell's *type families*, is used to define a more general type class of functors not limited to just *endofunctors* but allowing functors over *subcategories*, such as *Set* and other such restricted data types (Section 2).

Other commonly used categorical notions in Haskell built upon the concept of functors, such as *monads* and *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 *relative monads*, as shown by Altenkirch et al. [2, 3]. Section 3 applies the techniques of Section 2, defining classes of *relative monads* and their dual *relative comonads* (new in this paper) with examples. The functor and relative monad examples shown here appear elsewhere under the names *restricted functors* and *restricted monads* [4, 22]. Our contribution makes precise these constructions from a mathematical perspective.

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

Familiarity with Haskell up to type classes, and the basic concepts of *category*, *functor*, and *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 λ-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 $\bot$) and without advanced features such as a type classes, has a categorical semantics in terms of *bicartesian-closed* categories (*bccc*s) 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, σ and τ range over types, a ranges over type variables (potentially subscripted) and vector notation denotes multiple syntactic elements e.g. $\overline{a}$ is a group of type variables $a_0 \ldots a_{|a|}$.

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* $*$) as objects and Haskell functions as morphisms (any non-function expression is taken as a morphism from the *unit* value/type ()). The interpretation, given by $[\![-]\!]$, thus maps types to Hask_0 and functions to Hask_1 such that $[\![f :: \sigma \rightarrow \tau]\!] : [\![\sigma]\!] \rightarrow [\![\tau]\!] \in \mathsf{Hask}_1$ where $[\![\sigma]\!], [\![\tau]\!] \in \mathsf{Hask}_0$.

Note that Haskell does not provide an actual category 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 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 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:

$$\begin{aligned} [\![c]\!] &= c : \mathsf{Hask}_0 \\ [\![\sigma \rightarrow \tau]\!] &= \mathsf{Hask}([\![\sigma]\!], [\![\tau]\!]) : \mathsf{Hask}_0 && \textit{iff } [\![\sigma]\!] : \mathsf{Hask}_0 \wedge [\![\tau]\!] : \mathsf{Hask}_0 \\ [\![(\sigma, \tau)]\!] &= [\![\sigma]\!] \times [\![\tau]\!] : \mathsf{Hask}_0 && \textit{iff } [\![\sigma]\!] : \mathsf{Hask}_0 \wedge [\![\tau]\!] : \mathsf{Hask}_0 \end{aligned} \tag{1}$$

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 Hask are Haskell types, parametric polymorphism is central to categorical programming, providing universal quantification over types. Accordingly, only polymorphism over types of kind $*$ will be considered here.

The polymorphic λ-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*.[1] 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 $[\![-]\!]_{\Gamma}$ parameterised by a sequence Γ of indices for each free variable of the type it interprets (with some arbitrary canonical order). The definition of $[\![-]\!]_{\Gamma}$ is the same as $[\![-]\!]$ in (1), with Γ passed to recursive sub-terms, with an additional rule for type variables:

$$\begin{aligned} [\![a]\!]_{\Gamma} &= a : \mathsf{Hask}_0 && \textit{iff } a \in \Gamma \\ [\![\forall \overline{a} \,.\, \tau]\!]^{\forall} &= [\![\tau]\!] : (\textstyle\prod_{\overline{a}} \mathsf{Hask})_0 \mapsto \mathsf{Hask}_0 && \textit{iff } [\![\tau]\!]_{\overline{a}} : \mathsf{Hask}_0 \end{aligned} \tag{2}$$

[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 $(\prod_{\overline{a}} \mathsf{Hask})$ is the *product category* $(\mathsf{Hask} \times \ldots \times \mathsf{Hask})$ for the $|\overline{a}|$-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 $\overline{a} = \emptyset$ then $\prod_{\emptyset} \mathsf{Hask} = \mathsf{1}$, where $\mathsf{1}$ is the *unit category* with a single object and $\mathsf{1}_0 \mapsto \mathsf{Hask}_0 \cong \mathsf{Hask}_0$. Thus for a type with no type variables the polymorphic interpretation collapses to the monomorphic.

As an example, the polymorphic type of the *fst* function is interpreted:

$$[\![\forall a\, b \,.\, a \to (a, b)]\!]^{\forall}_{x,y} = \mathsf{Hask}(x, (x, y)) : (\mathsf{Hask} \times \mathsf{Hask})_0 \mapsto \mathsf{Hask}_0$$

Polymorphic functions are interpreted as indexed families of morphisms:

$$[\![f :: \forall \overline{a} \,.\, \sigma \to \tau]\!]^{\forall}_{\overline{a}} = [\![\forall \overline{a} \,.\, \sigma]\!]_{\overline{a}} \to [\![\forall \overline{a} \,.\, \tau]\!]_{\overline{a}} : (\textstyle\prod_{|\overline{a}|} \mathsf{Hask})_0 \mapsto \mathsf{Hask}_1 \qquad (3)$$

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

$$\begin{array}{rl} [\![F]\!]_{\Gamma} = F : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0 & \textit{iff} \;\; \textbf{data}\, F\; a \; = \ldots \\ [\![\sigma\, \tau]\!]_{\Gamma} = [\![\sigma]\!]_{\Gamma}\; [\![\tau]\!]_{\Gamma} : \mathsf{Hask}_0 & \textit{iff} \;\; [\![\sigma]\!]_{\Gamma} : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0 \;\wedge\; [\![\tau]\!]_{\Gamma} : \mathsf{Hask}_0 \end{array} \qquad (4)$$

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 $*$ and 2). *type constructors* of kind $* \to *$. 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, **class** $S \;\; a$ (where a has kind $*$), 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 $\mathsf{S}_0 \mapsto \mathsf{Hask}_0$ 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 $f : X \to Y \in \mathsf{S}_1$ then $X, Y \in \mathsf{S}_0$;*

– *for every morphism pair* $f : X \to Y, g : Y \to Z \in \mathsf{S}_1$ *then* $g \circ f : X \to Z \in \mathsf{S}_1$*;*
– *for every object* $X \in \mathsf{S}_0$ *there is an identity morphism* $id_X : X \to X \in \mathsf{S}_1$.

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 Hask is the upper bound. For example, the definition of *Ord* has a superclass constraint: **class** $Eq\ a \Rightarrow Ord\ a$ **where**.... thus any type that is an instance of *Ord* must be an instance of *Eq*, i.e., *Eq* is the *superclass* of *Ord* thus $\mathsf{Ord} \sqsubseteq \mathsf{Eq} \sqsubseteq \mathsf{Hask}$. Every subcategory S of C has an *inclusion functor* $I : \mathsf{S} \to \mathsf{C}$ mapping objects and morphisms into its *super*category.

A type with multiple constraints e.g. $\forall a \,.\, (Sa, Ta) \Rightarrow \tau$ will be interpreted as an $(\mathsf{S} \cap \mathsf{T})_0 \mapsto \mathsf{Hask}_0$ family, where $\mathsf{S} \cap \mathsf{T}$ 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 $[\![-]\!]_{\Delta|\Gamma}$ where $\Delta\,\tau$ is the set of type classes for which there are type class constraints over τ in the interpreted type. The definition of $[\![-]\!]_{\Delta|\Gamma}$ is given in Figure 1 where $(\bigcap_{S \in \Delta\tau} \mathsf{S})$ is the intersection subcategory $\mathsf{S}^0 \cap \ldots \cap \mathsf{S}^n$ for each class $S^i \in \Delta\tau$, i.e., the intersection category of all subcategories corresponding to class constraints on τ. If $\Delta\tau = \emptyset$ i.e. τ is unconstrained, then $\bigcap_{S \in \emptyset} \mathsf{S} = \mathsf{Hask}$. Thus for a type with no constraints, the ad-hoc polymorphic interpretation collapses to the polymorphic interpretation.

$$[\![a]\!]_{\Delta|\Gamma} = a : (\textstyle\bigcap_{S \in \Delta a}\ \mathsf{S})_0 \quad \textit{iff}\ a \in \Gamma \tag{5}$$
$$[\![c]\!]_{\Delta|\Gamma} = c : (\textstyle\bigcap_{S \in \Delta c}\ \mathsf{S})_0 \tag{6}$$
$$[\![(\sigma,\tau)]\!]_{\Delta|\Gamma} = [\![\sigma]\!] \times [\![\tau]\!] : (\textstyle\bigcap_{S \in \Delta(\sigma,\tau)}\ \mathsf{S})_0 \ \ \textit{iff}\ [\![\sigma]\!]_{\Delta|\Gamma} : \mathsf{C}_0 \wedge [\![\tau]\!]_{\Delta|\Gamma} : \mathsf{D}_0 \tag{7}$$
$$\begin{aligned}[\![\sigma \to \tau]\!]_{\Delta|\Gamma} &= (\mathsf{C} \sqcap \mathsf{D})([\![\sigma]\!]_{\Delta|\Gamma}, [\![\tau]\!]_{\Delta|\Gamma}) : (\textstyle\bigcap_{S \in \Delta(\sigma\to\tau)}\ \mathsf{S})_0 \\ &\qquad\qquad \textit{iff}\ [\![\sigma]\!]_{\Delta|\Gamma} : \mathsf{C}_0 \wedge [\![\tau]\!]_{\Delta|\Gamma} : \mathsf{D}_0\end{aligned} \tag{8}$$
$$\begin{aligned}[\![\sigma\,\tau]\!]_{\Delta|\Gamma} &= [\![\sigma]\!]_{\Delta|\Gamma}\ [\![\tau]\!]_{\Delta|\Gamma} : (\textstyle\bigcap_{S \in \Delta(\sigma\ \tau)}\ \mathsf{S})_0 \\ &\qquad\qquad \textit{iff}\ [\![\sigma]\!]_{\Delta|\Gamma} : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0 \ \wedge\ [\![\tau]\!]_{\Delta|\Gamma} : \mathsf{C}_0\end{aligned} \tag{9}$$

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 $\mathsf{Hask}_0 \mapsto \mathsf{Hask}_0$ family.

For tuples (7) and constructor applications (9), the subcategories of the subtypes σ and τ do not affect the subcategory of the constructed type since the constructors apply to any objects in Hask thus any objects of any Hask-subcategory.

For function types with source and target types in the same subcategory, the hom-object interpretation can be refined from 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 $\mathsf{C} \sqcap \mathsf{D}$, the *least upper bound category* as defined by $\sqsubseteq$ where $\mathsf{C} \sqcap \mathsf{C} = \mathsf{C}$. 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:

$$\begin{aligned}[\![f :: \forall \overline{a} \,.\, \overline{S\,\tau'} \Rightarrow \sigma \to \tau]\!]^{\forall\Rightarrow}_{\overline{a}} = [\![\forall \overline{a} \,.\, \overline{S\,\tau'} \Rightarrow \sigma]\!]_{\overline{a}} : \mathsf{C}_0 \to [\![\forall \overline{a} \,.\, \overline{S\,\tau'} \Rightarrow \tau]\!]_{\overline{a}} : \mathsf{D}_0 \\ : (\textstyle\prod_{a \in \overline{a}} \bigcap_{S \in \Delta a} \mathsf{S})_0 \mapsto (\mathsf{C} \sqcap \mathsf{D})_1 \qquad (10)\end{aligned}$$

2). Type classes parameterised by type constructors Type classes with a single parameter, e.g. **class** $F\ f$, where $f :: * \to *$ 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 $f : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0$ 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 *Functor* and restricted *map*-operations

Functor Since *Functor* is parameterised by a type constructor f of kind $* \to *$ it is interpreted as a meta definition, thus **class** *Functor* f **where** $\mathit{fmap} :: (a \to b) \to f\ a \to f\ b$ defines the *Functor*-structure comprising:

– an indexed family $f : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0$;
– an indexed family of morphisms *fmap*:

$$[\![\mathit{fmap}]\!]^{\forall\Rightarrow}_{a,b} = \mathsf{Hask}(a, b) \to \mathsf{Hask}(f\,a, f\,b) : (\mathsf{Hask} \times \mathsf{Hask})_0 \mapsto \mathsf{Hask}_1 \qquad (11)$$

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

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

with the usual functorial axioms. An *object mapping* is equivalent to an object-indexed family of objects, thus in the interpretation of *Functor*, the class parameter f matches the object mapping in the categorical definition where $C = D = \mathsf{Hask}$. The interpretation of *fmap* is a morphism in Hask over Hask hom-objects, analogous to the morphism mapping in the categorical definition but *embedded* within 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 $F_0 : C_0 \rightarrow C_0$ and an indexed family of morphisms:

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

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

Thus, *Functor* models *strong endofunctors* on Hask.

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

$$F.map :: (S\ \ a, S\ \ b) \Rightarrow (a \rightarrow b) \rightarrow F\ \ a \rightarrow F\ \ b$$

which is interpreted as $[\![F]\!]^{\forall\Rightarrow} : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0$ and:

$$[\![F.map]\!]^{\forall\Rightarrow}_{a,b} = \mathsf{S}(a,b) \rightarrow \mathsf{Hask}(F\,a, F\,b) : (\mathsf{S} \times \mathsf{S})_0 \mapsto \mathsf{Hask}_1 \tag{12}$$

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

The next section generalises *Functor* to non-endofunctors using recent features added to GHC/Haskell. The (non-standard) term *exofunctor* is used for emphasis to describe functors that need not be endofunctors.

2 Generalising *Functor* from endofunctors to *exo*functors

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

$$[\![exfmap]\!]^{\forall\Rightarrow}_{a,b} = \mathsf{S}(a,b) \rightarrow \mathsf{T}(f\,a, f\,b) : (S \times S)_0 \mapsto \mathsf{Hask}_1 \tag{13}$$

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 *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 redefining constraints as *types* with a distinct *constraint kind*, thus type families may return types of kind *Constraint*. The constraint kinds extension,[3] 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 $* \to \mathit{Constraint}$. Depending on the context, tuples can be types or constraints *i.e.* $(,) :: * \to * \to *$ or $(,) :: \mathit{Constraint} \to \mathit{Constraint} \to \mathit{Constraint}$ (conjunction of constraints) and $() :: *$ or $() :: \mathit{Constraint}$ 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]

```
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:

```
instance RFunctor Set where            instance RFunctor [ ] where
  type SubCats Set a = Ord a             type SubCats [ ] a = ()
  rfmap = Set.map                        rfmap = 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 $f\,a$ and $f\,b$, for both the source and target functions. For example:

```
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 *rfmap* as described by (13). For *Set*, the interpretation of *rfmap* is:

$$[\![\mathit{rfmap}]\!]^{\forall\Rightarrow}_{a,b} = \mathsf{Ord}(a,b) \to \mathsf{Hask}(\mathit{Set}\,a, \mathit{Set}\,b) : (\mathsf{Ord} \times \mathsf{Ord})_0 \mapsto \mathsf{Hask}_1 \qquad (14)$$

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*:

```
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:

```
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 $m : C_0 \to C_0$ [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:

```
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 *exo*functors may be *relative monads*, a generalisation of monads for functors $J : \mathsf{J} \to \mathsf{C}$ where J and C may be distinct [3].

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

- *a functor* $J : \mathsf{J} \to \mathsf{C}$
- *an object mapping* $T : \mathsf{J}_0 \to \mathsf{C}_0$
- *a natural transformation (unit)* $\eta_X : JX \to TX$ *(analogous to return)*
- *a natural transformation (extend)* $(-)^*_{X,Y} : (JX \to TY) \to (TX \to TY)$ *(analogous to* $\gg\!=$ *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:

```
class RMonad t where
  type RSubCats t x :: Constraint
```

$unit :: (RSubCats\ t\ x) \Rightarrow x \to t\ x$
$extend :: (RSubCats\ t\ x, RSubCats\ t\ y) \Rightarrow (x \to t\ y) \to t\ x \to t\ y$

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 $t : \mathsf{Hask}_0 \mapsto \mathsf{Hask}_0$ of *RMonad* corresponds to the object mapping T of relative monads, and is constrained to $t : \mathsf{J}_0 \mapsto \mathsf{C}_0$ 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 $I : \mathsf{J} \to \mathsf{C}$, thus $\mathsf{J} \sqsubseteq \mathsf{C}$. 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 $\mathsf{J} \sqsubseteq \mathsf{C}$ affects the interpretation of an *RMonad* instance in two ways depending on the relationship between J and C. Consider an instance of *RMonad*:

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 $\mathsf{J} \sqsubseteq \mathsf{C}$ therefore $\mathsf{J} \sqcap \mathsf{C} = \mathsf{C}$:

$$unit_a = a \to Foo\ a :: \mathsf{J}_0 \mapsto \mathsf{C}_1$$
$$extend_{a,b} = \mathsf{C}(a, Foo\ b) \to \mathsf{C}(Foo\ a, Foo\ b) : (\mathsf{J} \times \mathsf{J})_0 \mapsto \mathsf{Hask}_1$$

– if $\mathsf{J} \not\sqsubseteq \mathsf{C}$ therefore $\mathsf{J} \sqcap \mathsf{C} = \mathsf{Hask}$:

$$unit_a = a \to Foo\ a :: \mathsf{J}_0 \mapsto \mathsf{Hask}_1$$
$$extend_{a,b} = \mathsf{Hask}(a, Foo\ b) \to \mathsf{C}(Foo\ a, Foo\ b) : (\mathsf{J} \times \mathsf{J})_0 \mapsto \mathsf{Hask}_1$$

In the first case, the relative monad is over a functor $\mathsf{J} \to \mathsf{C}$ and in the second over a functor $\mathsf{J} \to \mathsf{Hask}$. 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 $\mathsf{J} = \mathsf{Ord}$ and $\mathsf{C} = \mathsf{Hask}$:

instance *RMonad Set* **where**
  type $RSubCats\ Set\ x = Ord\ x$
  $unit\ x = Set.singleton\ x$
  $extend\ f\ x = Set.unions\ (Prelude.map\ f\ (Set.toList\ x))$

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

instance $RMonad\ m \Rightarrow RFunctor\ m$ **where**
  type $SubCats\ m\ a = RSubCats\ m\ a$
  $rfmap\ f = extend\ (unit \circ f)$

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:

$$\begin{array}{l}
\textbf{class}\ \mathit{Comonad}\ c\ \textbf{where} \\
\quad \mathit{coreturn} :: c\ a \to a \\
\quad (\ggg) :: c\ a \to (c\ a \to b) \to c\ b
\end{array}$$

with dual laws to those of monads [23, 24], where $\ggg$ 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 $c\ a \to b$ 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:

$$\begin{array}{l}
\textbf{data}\ \mathit{Arr}\ i\ a = \mathit{Arr}\ (\mathit{Array}\ i\ a)\ i \\
\textbf{instance}\ \mathit{Ix}\ i \Rightarrow \mathit{Comonad}\ (\mathit{Arr}\ i)\ \textbf{where} \\
\quad \mathit{coreturn}\ (\mathit{Arr}\ \mathit{arr}\ c) = \mathit{arr}\ !\ c \\
\quad (\mathit{Arr}\ x\ c)\ \ggg f = \textbf{let}\ \mathit{es}' = \mathit{map}\ (\lambda i \to (i, f\ (\mathit{Arr}\ x\ i)))\ (\mathit{indices}\ x) \\
\qquad\qquad\qquad\qquad\ \ \textbf{in}\ \mathit{Arr}\ (\mathit{array}\ (\mathit{bounds}\ x)\ \mathit{es}')\ c
\end{array}$$

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

$$\begin{array}{l}
\mathit{laplace1D}\ (\mathit{Arr}\ a\ i) = \textbf{if}\ (i > 0 \wedge i < (n-1)) \\
\qquad\qquad\qquad\qquad\quad \textbf{then}\ a\ !\ (i-1) - 2 * (a\ !\ i) + a\ !\ (i+1)\ \textbf{else}\ 0.0 \\
n = \mathit{length}\ \mathit{inputData} \\
x = \mathit{Arr}\ (\mathit{array}\ (0, n)\ \mathit{inputData})\ 0 \\
x' = x\ \ggg \mathit{laplace1D}
\end{array}$$

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

$$(!) :: (\mathit{IArray}\ a\ e, \mathit{Ix}\ i) \Rightarrow a\ i\ e \to i \to e$$

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 $\mathsf{Hask} \to \mathsf{Hask}$ 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. *UArray* is thus restricted and is an exofunctor (IArray UArray) $\rightarrow$ Hask, therefore cannot be a comonad. However, as with monads, comonads can be generalised to *relative comonads* on exofunctors.

Definition 3. *A* relative comonad *dualises a* relative monad*, and is defined over categories* K *and* C*, comprising:*

- *a functor* $K : \mathsf{K} \rightarrow \mathsf{C}$
- *an object mapping* $D : \mathsf{K}_0 \rightarrow \mathsf{C}_0$
- *a natural transformation (counit):* $\epsilon_X : DX \rightarrow KX$
- *a natural transformation (coextend):* $(-)^{\dagger}_{X,Y} : (DX \rightarrow KY) \rightarrow (DX \rightarrow DY)$

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

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

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

As with relative monads, the functor K in the categorical definition is taken as the inclusion functor $\mathsf{K} \rightarrow \mathsf{C}$ 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:

```
data UArr i a = UArr (UArray i a) i
instance Ix i ⇒ RComonad (UArr i) where
  type RCSubCats (UArr i) x = IArray UArray x
  counit (UArr arr c) = ...       -- same as coreturn for the Arr comonad
  coextend f (UArr x c) = ...     -- same as (⇛) for the Arr comonad
```

As another example, the notion of a *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 *Set* data type:

```
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 (λa' → f (PSet (Set.delete a' s) a')) s) (f (PSet s a))
```

where *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 *RCSubCats* 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:

```
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 *RObjs* and *RMorphs*. Since *counit* involves just objects of K and C the single parameter *RObjs* is used, specifying objects of the subcategories. However, for *coextend*, involving objects *and* morphisms of J and C, *RMorphs* 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]:

```
    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 $r :: [a] \to [a]$ and any $f :: x \to y$:

$$r \circ (map\ f) = (map\ f) \circ r$$

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

$$r \circ (fmap\ f) = (fmap\ f) \circ r$$

The occurrence of *fmap* on the left is for the data type G and that on the right for the data type F and r is instantiated at different types on the left- and right-hand sides. This theorem corresponds to the *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, $\mathit{sort} :: \mathit{Ord}\ a \Rightarrow [a] \to [a]$ has the free theorem, for any $f :: a \to b$:

$$(\mathit{fmap}\ f) \circ \mathit{sort} = \mathit{sort} \circ (\mathit{fmap}\ f) \ \ \mathit{iff}\ \ \forall x, y :: a\,.x \leqslant y \Rightarrow (f\,x) \leqslant (f\,y)$$

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

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

$$\begin{array}{l} \text{Given an operation } r :: \mathit{Set}\ a \to \mathit{Set}\ a \quad \text{then } \forall f :: (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow a \to b \\ \quad r \circ (\mathit{fmap}\ f) = (\mathit{fmap}\ f) \circ r \quad \mathit{iff}\ \ \forall x, y :: a\,.\,x \leqslant y \Rightarrow (f\,x) \leqslant (f\,y) \end{array}$$

For example, the operation $\mathit{deleteMin} :: \mathit{Set}\ a \to \mathit{Set}\ a$ is only natural with respect to monotonic functions since it does not compare elements itself, but simply uses the internal binary tree structure of *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 $r :: (\mathit{Functor}\ f) \Rightarrow \mathit{Set}\ a \to f\ a$ and $r :: (\mathit{Functor}\ f) \Rightarrow f\ a \to \mathit{Set}\ a$ the same free theorem as above holds, plus any additional restrictions to naturality provided by the functor f.

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 $\mathbf{S}$ and thus are functors $\mathbf{S} \to \mathbf{ADT}$ (where $\mathbf{ADT}$ 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., ($\mathbf{data}\ \mathit{Ord}\ a \Rightarrow \mathit{Set}\ a = ...$). Consequently, the *Ord* constraint need not appear in the types of operations, e.g., $\mathit{Set.map} :: (a \to b) \to \mathit{Set}\ a \to \mathit{Set}\ b$. 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 *functor* and *monad* classes, requiring manual encoding and passing of constraints using GADTs. The names *RFunctor* and *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 λ-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

1. Adams, S. Functional Pearls: Efficient sets–a balancing act. *Journal of Functional Programming 3*, 4 (1993), 553–562.
2. Altenkirch, T., Chapman, J., and Uustalu, T. Relative monads formalised. *To appear in the Journal of Formalized Reasoning. Final version pending*.
3. Altenkirch, T., Chapman, J., and Uustalu, T. Monads need not be endofunctors. *Foundations of Software Science and Computational Structures* (2010), 297–311.
4. Bolingbroke, M. Constraint Kinds for GHC, 2011. `http://blog.omega-prime.co.uk/?p=127` (Retreived 14/09/11).
5. Chakravarty, M. M. T., Keller, G., and Jones, S. P. Associated type synonyms. In *ICFP '05* (2005), ACM, pp. 241–253.
6. Danielsson, N., Hughes, J., Jansson, P., and Gibbons, J. Fast and loose reasoning is morally correct. In *POPL '06* (2006), ACM, pp. 206–217.
7. Fokkinga, M. *A Gentle Introduction to Category Theory-the calculational approach.* University of Utrecht, 1992.
8. Gunter, C. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. Mit Press, 1992.
9. Hughes, J. Restricted Data Types in Haskell. In *Proceedings of the 1999 Haskell Workshop. Technical Report UU-CS-1999-28* (Utrecht, 1999).
10. Jones, M. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (1993), ACM, pp. 52–61.
11. Kelly, G. *Basic concepts of enriched category theory*, vol. 64. Cambridge Univ Pr, 1982.

12. Kieburtz, R. B. Codata and Comonads in Haskell, 1999.
13. Kock, A. Strong functors and monoidal monads. *Archiv der Mathematik 23*, 1 (1972), 113–120.
14. Lambek, J., and Scott, P. *Introduction to higher order categorical logic.* Cambridge University Press, 1988.
15. Manes, E. *Algebraic theories.* Springer, 1976.
16. Nogueira, P. When is an abstract data type a functor? *Trends in Functional Programming 7* (2007), 217–231.
17. Orchard, D., Bolingbroke, M., and Mycroft, A. Ypnos: Declarative, Parallel Structured Grid Programming. In *DAMP '10: Proceedings of workshop on Declarative aspects of multicore programming* (NY, USA, 2010), ACM, pp. 15–24.
18. Orchard, D., and Schrijvers, T. Haskell Type Constraints Unleashed. *Functional and Logic Programming* (2010), 56–71.
19. Peter, J. d. B. Naturalness of polymorphism. Tech. rep., Technical Report CS 8916, Rijksuniversiteit Groningen, The Netherlands, 1989.
20. Reynolds, J. Types, abstraction and parametric polymorphism. In *Information Processing* (1983), R. Mason, Ed., Elsevier Science Publishers B.V.
21. Seely, R. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic* (1987), 969–989.
22. Sittampalam, G., and Gavin, P. rmonad: Restricted monad library, 2008. `http://hackage.haskell.org/package/rmonad`.
23. Uustalu, T., and Vene, V. Comonadic Notions of Computation. *Electron. Notes Theor. Comput. Sci. 203*, 5 (2008), 263–284.
24. Uustalu, T., and Vene, V. The Essence of Dataflow Programming. *Lecture Notes in Computer Science 4164* (Nov 2006), 135–167.
25. Wadler, P. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture* (1989), ACM, pp. 347–359.
26. Wadler, P. Monads for functional programming. *Advanced Functional Programming* (1995), 24–52.