Subcategories & “Exofunctors” in Haskell

dorchard haskell, maths October 18, 2011July 24, 2012 7 Minutes

In my previous post (https://dorchard.wordpress.com/2011/09/22/constraint-kinds-in-haskell-finally-bringing-us-constraint-families/) I discussed the new *constraint kinds* extension to GHC, which provides a way to get type-indexed constraint families in GHC/Haskell. The extension provides some very useful expressivity. In this post I’m going to explain a possible use of the extension.

In Haskell the `Functor` class is misleading named as it actually captures the notion of an *endo*functor, not functors in general. This post shows a use of constraint kinds to define a type class of *exo*functors; that is, functors that are not necessarily endofunctors. I will explain what all of this means.

This example is just one from a draft note (edit July 2012: draft note subsumed by my TFP 2012 submission (http://www.cl.cam.ac.uk/~dao29/drafts/tfp-structures-orchard12.pdf)) explaining the use of constraint families, via the constraint kinds extension, for describing abstract structures from category theory that are parameterised by *subcategories*, including non-endofunctors, *relative monads*, and *relative comonads*.

I will try to concisely describe any relevant concepts from category theory, through the lens of functional programming, although I’ll elide some details.

The **Hask** category

The starting point of the idea is that programs in Haskell can be understood as providing definitions within some category, which we will call **Hask**. Categories comprise a collection of *objects* and a collection of *morphisms* which are mappings between objects. Categories come equipped with identity morphisms for every object and an associative composition operation for morphisms (see Wikipedia (http://en.wikipedia.org/wiki/Category_theory#Categories.2C_objects.2C_and_morphisms) for a more complete, formal definition). For **Hask**, the objects are Haskell types, morphisms are functions in Haskell, identity morphisms are provided by the identity function, and composition is the usual function composition operation. For the purpose of this discussion we are not really concerned about the exact properties of **Hask**, just that Haskell acts as a kind of *internal language* for category theory, within some arbitrary category **Hask** (Dan Piponi provides some discussion on this topic (http://blog.sigfpe.com/2009/10/what-category-do-haskell-types-and.html)).

Subcategories

Given some category *C*, a *subcategory* (http://en.wikipedia.org/wiki/Subcategory) of *C* comprises a subcollection of the objects of *C* and a subcollection of the morphisms of *C* which map only between objects in the subcollection of this subcategory.

We can define for **Hask** a *singleton* subcategory for each type, which has just that one type as an object and functions from that type to itself as morphisms e.g. the *Int*-subcategory of **Hask** has one object, the `Int` type, and has functions of type `Int → Int` as morphisms. If this subcategory has *all* the morphisms `Int → Int` it is called a *full* subcategory. Is there a way to describe "larger" subcategories with more than just one object?

Via universal quantification we could define the trivial ("non-proper" (http://en.wikipedia.org/wiki/Proper_subset#proper_subset)) subcategory of **Hask** with objects of type `a` (implicitly universally quantified) and morphisms `a -> b`, which is just **Hask** again. Is there a way to describe "smaller" subcategories with fewer than all the objects, but more than one object? Yes. For this we use type classes.

Subcategories as type classes

The instances of a single parameter type class can be interpreted as describing the members of a set of types (or a relation on types for multi-parameter type classes). In a type signature, a universally quantified type variable constrained by a type class constraint represents a collection of types that are members of the class. E.g. for the `Eq` class, the following type signature describes a collection of types for which there are instances of `Eq`:

```
Eq a => a
```

The members of `Eq` are a subcollection of the objects of **Hask**. Similarly, the type:

```
(Eq a, Eq b) => (a -> b)
```

represents a subcollection of the morphisms of **Hask** mapping between objects in the subcollection of objects which are members of `Eq`. Thus, the `Eq` class defines an *Eq*-subcategory of **Hask** with the above subcollections of objects and morphisms.

Type classes can thus be interpreted as describing subcategories in Haskell. In a type signature, a type class constraint on a type variable thus specifies the subcategory which the type variable ranges over the objects of. We will go on to use the constraint kinds extension to define constraint-kinded type families, allowing structures from category theory to be parameterised by subcategories, encoded as type class constraints. We will use *functors* as the example in this post (more examples here (http://www.cl.cam.ac.uk/~dao29/drafts/subcategories-in-haskell-dorchard11.pdf)).

Functors in Haskell

In category theory, a functor provides a mapping between categories e.g. $F : C \rightarrow D$, mapping the objects and morphisms of C to objects and morphisms of D. Functors preserves identities and composition between the source and target category (see Wikipedia (http://en.wikipedia.org/wiki/Functor) for more). An *endofunctor* is a functor where C and D are the same category.

The type constructor of a parametric data type in Haskell provides an object mapping from **Hask** to **Hask** e.g. given a data type `data F a = ...` the type constructor `F` maps objects (types) of **Hask** to other objects in **Hask**. A functor in Haskell is defined by a parametric data type, providing an object mapping, and an instance of the well-known `Functor` class for that data type:

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

which provides a mapping on morphisms, called `fmap`. There are many examples of functors in Haskell, for examples lists, where the `fmap` operation is the usual map operation, or the `Maybe` type. However, not all parametric data types are functors.

It is well-known that the `Set` data type in Haskell cannot be made an instance of the `Functor` class. The `Data.Set` library provides a map operation of type:

```
Set.map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
```

The `Ord` constraint on the element types is due to the implementation of `Set` using balanced binary trees, thus elements must be comparable. Whilst the data type is declared polymorphic, the constructors and transformers of `Set` allow only elements of a type that is an instance of `Ord`.

Using `Set.map` to define an instance of the `Functor` class for `Set` causes a type error:

```
instance Functor Set where
   fmap = Data.Set.map

...

foo.lhs:4:14:
    No instances for (Ord b, Ord a)
      arising from a use of `Data.Set.map'
    In the expression: Data.Set.map
    In an equation for `fmap': fmap = Data.Set.map
    In the instance declaration for `Functor Set'
```

The type error occurs as the signature for `fmap` has no constraints, or the *empty* (always true) constraint, whereas `Set.map` has `Ord` constraints. A mismatch occurs and a type error is produced.

The type error is however well justified from a mathematical perspective.

Haskell functors are not functors, but endofunctors

First of all, the name `Functor` is a misnomer; the class actually describes *endo*functors, that is functors which have the same category for their source and target. If we understand type class constraints as specifying a subcategory, then the lack of constraints on `fmap` means that `Functor` describes endofunctors **Hask** → **Hask**.

The `Set` data type is not an endofunctor; it is a functor which maps from the *Ord*-subcategory of **Hask** to **Hask**. Thus `Set :: Ord → Hask`. The class constraints on the element types in `Set.map` declare the subcategory of `Set` functor to which the morphisms belong.

Type class of *exo*functors

Can we define a type class which captures functors that are not necessarily endofunctors, but may have distinct source and target categories? Yes, using an associated type family of kind `Constraint` .

The following `ExoFunctor` type class describes a functor from a subcategory of **Hask** to **Hask**:

```
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE TypeFamilies #-}

class ExoFunctor f where
   type SubCat f x :: Constraint
   fmap :: (SubCat f a, SubCat f b) => (a -> b) -> f a -> f b
```

The `SubCat` family defines the source subcategory for the functor, which depends on `f`. The target subcategory is just **Hask**, since `f a` and `f b` do not have any constraints.

We can now define the following instance for `Set`:

```
instance ExoFunctor Set where
   type SubCat Set x = Ord x
   fmap = Set.map
```

Endofunctors can also be made an instance of `ExoFunctor` using the empty constraint e.g.:

```
instance ExoFunctor [] where
    type SubCat [] a = ()
    fmap = map
```

(Aside: one might be wondering whether we should also have a way to restrict the target subcategory to something other than **Hask** here. By covariance we can always "cast" a functor $C \to D$, where D is a subcategory of some other category E, to $C \to E$ without any problems. Thus, there is nothing to be gained from restricting the target to a subcategory, as it can always be reinterpreted as **Hask**.)

Conclusion (implementational restrictions = subcategories)

Subcategory constraints are needed when a data type is restricted in its polymorphism by its operations, usually because of some hidden implementational details that have permeated to the surface. These implementational details have until now been painful for Haskell programmers, and have threatened abstractions such as functors, monads, and comonads. Categorically, these implementational restrictions can be formulated succinctly with subcategories, for which there are corresponding structures of non-endofunctors, relative monads, and relative comonads. Until now there has been no succinct way to describe such structures in Haskell.

Using constraint kinds we can define associated type families, of kind `Constraint`, which allow abstract categorical structures, described via their operations as a type class, to be parameterised by subcategories on a per-instance basis. We can thus define a class of *exo*functors, i.e. functors that are not necessarily endofunctors, which we showed here. The other related structures which are difficult to describe in Haskell without constraint kinds: *relative monads* and *relative comonads*, are discussed further in a draft note (edit July 2012: draft note subsumed by my TFP 2012 submission (http://www.cl.cam.ac.uk/~dao29/drafts/tfp-structures-orchard12.pdf)). The note includes examples of a *Set* monad and an unboxed array comonad, both of which expose their implementational restrictions as type class constraints which can be described as subcategory constraints.
Any feedback on this post or the draft note is greatly appreciated. Thanks.

Tagged:
category theory,
constraints,
Haskell,
subcategories

Published by dorchard

View all posts by dorchard

10 thoughts on "Subcategories & "Exofunctors" in Haskell"

Steven says:

October 18, 2011 at 4:01 pm
Doe this mean that in some language Hask could be realized as a type class?
Something like

class Hask a where
y::(a->a)->a
id::a->a

Of course it'd be a lot bigger and broken up into very basic classes.
This would allow for alternate classes instead of merely subcategories of Hask.
Who knows maybe even uniqueness types could be implemented this way.

↳ Reply
dorchard says:
October 19, 2011 at 11:27 am
Hi Steven, I'm not quite sure if I follow exactly what you are saying.
Following the ideas in the article, Hask can be described:

```
class Hask a
instance Hask a
```

I'm not sure if I follow what your intention with "y" is. Categories can be encoded "internally" in Haskell with something like:

```
class Category c where
id :: c a a
comp :: c x y -> c y z -> c x z
```

which might be something along the lines of what you are thinking.

Using this internal representation of categories, functors could be defined as:

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

Is this roughly what you had in mind?
Dominic

↳ Reply
bmeph says:
October 23, 2011 at 9:11 am
Wouldn't that be more like:

class (Category c, Category d) => Functor f c d where
fmap :: c a b -> d (f a) (f b)
?

↳ Reply
dorchard says:
October 23, 2011 at 3:37 pm
Right. I forgot the constraints there.

↳ Reply
Gabor Greif says:
July 25, 2012 at 12:30 am
On page 4 of the paper (bottom) you give a strange type to the fst function: fst :: forall a b . a -> (a,b). This cannot work. Also the Hask(x, (x,y)) should probably be Hask((x,y), x).

↳ Reply
dorchard says:
July 25, 2012 at 9:39 am
Gabor: thanks for spotting the mistake. I somehow flipped the type! I have fixed the online version.

Apologies to Planet Haskell – apparently when I edit an old post it appears again on the Planet Haskell feed (even though the date of this post is in the past).

↳ Reply
Edward Kmett says:
October 13, 2013 at 6:48 am
Note: instance ExoFunctor Set where type SubCat Set x = Ord x can only pass the laws if you presume a lot about Ord, (well actually, even just Eq), that isn't satisfied for many instances:

Notably Eq lacks a claim that x == y entails f x = f y as neither Eq nor Ord claims to be structural.

Sadly, even reflexivity fails in practice. 😉

↳ Reply
Anibal says:
October 26, 2013 at 8:54 pm
Hi there every one, here every one is sharing such knowledge, therefore it's pleasant to read this weblog, and I used to pay a visit this blog every day.

↳ Reply
Pingback: Is there a library that uses ConstraintKinds to generalize all the base type classes to allow constraints? | Ask Programming & Technology
A says:
September 19, 2015 at 9:33 am
Hello, nice blog post! I was looking for something like this when google searched up your blog post.

I think there is something to be gained from being able to restrict the image subcategory.

Suppose I have a map f:a -> b and a map g:b -> c, and moreover, there are canonical injections of everything into some d. Of course I can get maps f':a->d and g': b->d by composing f and g with the injections. However, something is lost in this translation: the maps f and g naturally compose, but f' and g' do not: they don't `typecheck'.

For example, if C were some subcategory of Hask, and f: C->C were an endofunctor of C, it is not clear how one would compose f with itself if all we remembered was the map f': C -> Hask.

(This is silly and obvious, but I would like to point it out because I needed to compose my functors, so I disagreed with the statement that keeping track of the image was unneeded).

↳ Reply