Reverse Engineering Machines with the Yoneda Lemma

Dan Piponi

February 2024

1 Introduction

When I wrote about coends a while back I made up a term 'difunctor'. More recently it was pointed out to me that the correct word for this concept is 'profunctor', but unfortunately my knowledge came from MacLane which mentions that word nowhere.

Profunctors are ubiquitous in Haskell programming. Probably the most natural definition of Hughes Arrows is via profunctors. Profunctors also play a role a little like tensors leading to a use of the terms 'covariant' and 'contravariant' that looks remarkably like the way those terms are used in tensor calculus.

For categories C and D, A profunctor is a functor $D^{op} \times C \to Set$ and is written $C \nrightarrow D$. (I hope that arrow between C and D is in your font. It's missing on iOS.)

I'll reuse my Haskell approximation to that definition:

```
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE ExistentialQuantification #-}

class Profunctor h where
  lmap :: (d' -> d) -> h d c -> h d' c
  rmap :: (c -> c') -> h d c -> h d c'
```

We need cofunctoriality for the first argument and functoriality for the second:

```
lmap (f . g) == lmap g . lmap f
rmap (f . g) == rmap f . rmap g
```

(Strictly we probably ought to call these 'endoprofunctors' as we're only really dealing with the category of Haskell types and functions.)

There are lots of analogies for thinking about profunctors. For example, some people think of them as generalising functors in the same way that relations generalise functions. More specifically, given a function $f : A \to B$, f associates to each element of A, a single element of B. But if we want f to associate elements of A with elements of B more freely, for example 'mapping' elements of A to multiple elements of B then we instead use a relation which can be written as a function $f : A \times B \to \{0, 1\}$ where we say xfy iff $f(x, y) = 1$. In this case, profunctors map to Set rather than $\{0, 1\}$.

A good example is the type constructor `(->)`

```
instance Profunctor (->) where
  lmap f g = g . f
  rmap f g = f . g
```

It's common that the first argument of a profunctor describes how an element related to a type is sucked in, and the second describes what is spit out. `a -> b` sucks in an `a` and spits out a `b`.

Given a function f we can turn it into a relation by saying that xfy iff $y = f(x)$. Similarly we can turn a functor into a profunctor. Given a functor $F : C \to D$ we can define a profunctor $F^* : C \nrightarrow D$ by

```
data UpStar f d c = UpStar (d -> f c)
instance Functor f => Profunctor (UpStar f) where
  lmap k (UpStar f) = UpStar (f . k)
  rmap k (UpStar f) = UpStar (fmap k . f)
```

You may be able to see how the second argument to a profunctor sort of plays a similar role to the return value of a functor, just as the second argument to a relation sometimes plays a rule similar to the return value of a function.

There also an opoosing way to make a profunctor from a functor just as there is with functions and relations:

```
data DownStar f d c = DownStar (f d -> c)
instance Functor f => Profunctor (DownStar f) where
  lmap k (DownStar f) = DownStar (f . fmap k)
  rmap k (DownStar f) = DownStar (k . f)
```

Note that the identity functor gives us something isomorphic to `(->)` whether you use `UpStar` or `DownStar`.

2 Dinatural transformations

Just as we have natural transformations between functors, we have dinatural transformations between profunctors. My previous definition of dinatural was specialised to a particular case - dinaturals between a profunctor and the constant profunctor.

Firstly, let's think about natural transformations. If F and G are functors, and h is a natural transformation $h : F \implies G$, then we have that

```
h . fmap f = fmap f . h
```

If we think of F and G as containers, then this rule says that a natural transformation relates the structures of the containers, not the contents. So using f to replace the elements with other elements should be invisible to h and hence commute with it.

Something similar happens with dinatural transformations. But this time, instead of relating the argument to a natural transformation to its return result, it instead relates the two arguments to a profunctor.

Given two profunctors, F and G, A dinatural transformation is a polymorphic function of type:

```
type Dinatural f g = forall a. f a a -> g a a
```

but we also want something analogous to the case of natural transformations. We want to express the fact that if `phi :: Dinatural F G`, then `phi`

doesn't see the elements of `F a a` or `G a a`. Here's a way to achieve this. Suppose we have a dinatural transformation:

```
phi :: Dinatural G F
```

and a function `f :: X -> X'` then we can use `lmap` to apply `f` on the left or right of `F` and `G`. The definition of dinaturals demands that:

```
rmap f . phi . lmap f = lmap f . phi . rmap f
```

ie. that we can apply `f` on the left before applying `phi`, and then do `f` on the right, or vice versa, and still get the same result.

I'm not sure but I think that we don't need to check this condition and that just like the case of naturals it just comes as a free theorem.

3 Composing profunctors

It's easy to see how to compose functors. A functor is a polymorphic function from one type to another. It's not straightforward to compose profunctors. It's tempting to say that a profunctor maps a pair of types to a type so they can be composed like functions. But the original definition says the definition is $D^{op} \times C \to Set$. So as a function it doesn't map back to the category but to Set. For Haskell we replace Set with $Hask$, the category of Haskell functions and types. So we have $Hask^{op} \times Hask \to Hask$. It's easy invent a scheme to compose these because $Hask$ appears 3 times. But it'd be wrong to exploit this in a general definition applying to many categories because in the proper definition of profunctor we can't assume that a profunctor maps back to the spaces you started with.

We can try composing profunctors by analogy with composing relations. Suppose R and S are relations. If $T = S \circ R$ is the composition of R and S then xTz if and only if there exists a y such that xRy and ySz. If our relations are on finite sets then we can define $T(x,z) = \sum_y R(x,y)S(y,z)$ where we work in the semiring on $\{0,1\}$ with $0+0=0$, $0+1=1+0=1+1=1$ but with the usual product.

There is an analogue of "there exists" in Haskell - the existential type. Remembering that we write Haskell existential types using `forall` we

can define:

```
data Compose f g d c = forall a.Compose (f d a) (g a c)
```

As mentioned above, functors give rise to profunctors. It'd be good if composition of functors were compatible with composition of profunctors. So consider

```
Compose (UpStar F) (UpStar G)
```

for some `F` and `G`. This is essentially the same as

```
exists a. (d -> F a, a -> G c)
```

What can we discover about an element of such a type? It consists of a pair of functions `(f, g)`, but we can't ever extract the individual functions because the type of `a` has been erased. To get anything meaningful out of `g` we need to apply it to an `a`, but we don't have one immediately to hand, after all, we can't even know what `a` is. But we do have an `F a` if we can make a `d`. So we can use `fmap` to apply `g` to the result of `a`. So we can construct `fmap g . f :: d -> F (G c)`. There is no other information we can obtain. So the composition is isomorphic to `UpStar` of the functorial composition of `F` and `G`. Again, we can probably make this a rigorous proof by making use of free theorems, but I haven't figured that out yet.

But there's a catch: I said I wanted a definition that applies to more categories than just Hask. Well we can replace `exists a` with the coend operator. We also implicitly used the product operation in the constructor `Compose` so this definition will work in categories with suitable products. Symmetric monodial categories in fact.

Under composition of profunctors, `(->)` is the identity. At least up to isomorphism. This composition of profunctors is also associative up to isomorphism. Unfortunately the "up to isomorphism" means that we can't make a category out of profunctors in the obvious way. But we can make a bicategory - essentially a category where we have to explicitly track the isomorphisms between things that are equal in ordinary categories.

4 Profunctors as tensors

Given a profunctor `F` we can write `F i j` suggestively as F_i^j. Let's write the composition of `F` and `G` as $\exists k.F_i^k G_k^j$. We can use Einstein notation to automatically 'contract' on pairs of upper and lower indices and write the composition as $F_i^k G_k^j$. The analogy is even more intriguing when we remember that in tensor notation, the upper indices are covariant indices and the lower ones are contravariant indices. In the case of profunctors, the two arguments act like the arguments to covariant and contravariant functors respectively. Note alse that because `(->)` is essentially the identity, we have $\to_i^j F_j^k = F_i^k$. So `(->)` acts like the Kronecker delta. You can read more about this at mathoverflow where it is hinted that this analogy is not yet well understood. Note that we're naturally led to the trace of a profunctor: `exists a. F a a`.

5 Arrows as profunctors

The last thing I want to mention is that Hughes' Arrows are profunctors. There is an intuition that fits. If `A` is an Arrow, we often think of `A d c` as consuming something related to type `d` and emitting something related to type `c`. The same goes for profunctors. The full paper explaining this is Asada and Hasuo's Categorifying Computations into Components via Arrows as Profunctors

6 Afterword

Profunctors are now a standard part of Haskell in Data.Profunctor. `UpStar` was renamed to `Upstar` and `DownStar` was renamed to `Costar`. *Costar* is also the name of an astrology company that uses Haskell.

Profunctors form the basis for profunctor optics. See Profunctor Optics: Modular Data Accessors by Matthew Pickering, Jeremy Gibbons and Nicolas Wu.