Bartosz Milewski's Programming Cafe

Category Theory, Haskell, Concurrency, C++

Category Theory

Archived Posts from this Category

September 20, 2019

The Power of Adjunctions

Posted by Bartosz Milewski under Category Theory, Monads, Programming
[15] Comments

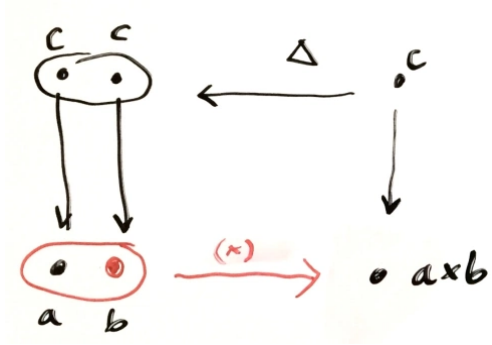
i
6 Votes

In my previous blog post, Programming with Universal Constructions, I mentioned in passing that one-to-one mappings between sets of morphisms are often a manifestation of adjunctions between functors. In fact, an adjunction just extends a universal construction over the whole category (or two categories, in general). It combines the mapping-in with the mapping-out conditions. One functor (traditionally called the left adjoint) prepares the input for mapping out, and the other (the right adjoint) prepares the output for mapping in. The trick is to find a pair of functors that complement each other: there are as many mapping-outs from one functor as there are mapping-ins to the other functor.

To gain some insight, let's dig deeper into the examples from my previous post.

The defining property of a *product* was the universal mapping-in condition. For every object c equipped with a pair of morphisms going to, respectively, a and b, there was a unique morphism h mapping c to the product $a \times b$. The commuting condition ensured that the correspondence went both ways, that is, given an h, the two morphisms were uniquely determined.

A pair of morphisms can be seen as a single morphism in a product category $C \times C$. So, really, we have an isomorphism between hom-sets in two categories, one in $C \times C$ and the other in C. We can also define two functors going between these categories. An arbitrary object c in C is mapped by the diagonal functor Δ to a pair $\langle c, c \rangle$ in $C \times C$. That's our left functor. It prepares the source for mapping out. The right functor maps an arbitrary pair $\langle a, b \rangle$ to the product $a \times b$ in C. That's our target for mapping in.



The adjunction is the (natural) isomorphism of the two hom-sets:

$$(C \times C)(\Delta c, \langle a, b \rangle) \cong C(c, a \times b)$$

Let's develop this intuition. As usual in category theory, an object is defined by its morphisms. A product is defined by the mapping-in property, the totality of morphisms incoming from all other objects. Hence we are interested in the hom-set between an arbitrary object c and our product $a \times b$. This is the right hand side of the picture. On the left, we are considering the mapping-out morphism from the object $\langle c, c \rangle$, which is the result of applying the functor Δ to c. Thus an adjunction relates objects that are defined by their mapping-in property and objects defined by their mapping-out property.

Another way of looking at the pair of adjoint functors is to see them as being approximately the inverse of each other. Of course, they can't, because the two categories in question are not isomorphic. Intuitively, $C \times C$ is "much bigger" than C. The functor that assigns the product $a \times b$ to every pair $\langle a, b \rangle$ cannot be injective. It must map many different pairs to the same (up to isomorphism) product. In the process, it "forgets" some of the information, just like the number 12 forgets whether it was obtained by multiplying 2 and 6 or 3 and 4. Common examples of this forgetfulness are isomorphisms such as

$$a \times b \cong b \times a$$

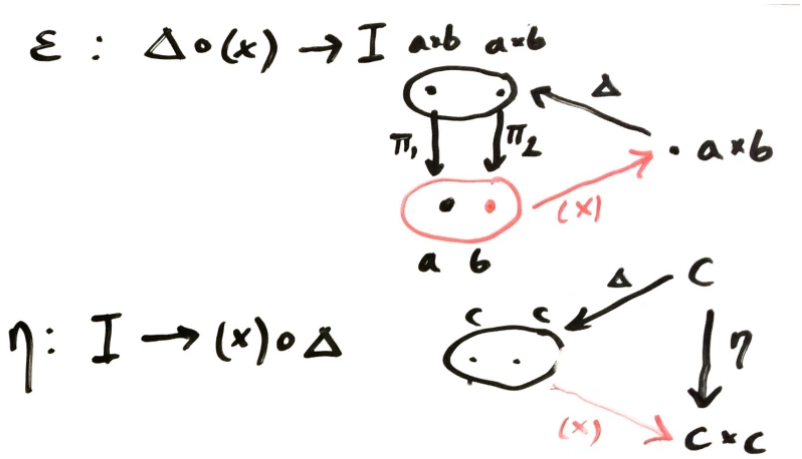
or

$$(a \times b) \times c \cong a \times (b \times c)$$

Since the product functor loses some information, its left adjoint must somehow compensate for it, essentially by making stuff up. Because the adjunction is a natural transformation, it must do it uniformly across the whole category. Given a generic object c, the only way it can produce a pair of objects is to duplicate c. Hence the diagonal functor Δ. You might say that Δ "freely" generates a pair. In almost every adjunction you can observe this interplay of "freeness" and "forgetfulness." I'm using these therm loosely, but I can be excused, because there is no formal definition of forgetful (and therefore free or cofree) functors.

Left adjoints often create free stuff. The mnemonic is that "the left" is "liberal." Right adjoints, on the other hand, are "conservative." They only use as much data as is strictly necessary and not an iota more (they also preserve limits, which the left adjoints are free to ignore). This is all relative and, as we'll see later, the same functor may be the left adjoint to one functor and the right adjoint to another.

Because of this lossiness, a round trip using both functors doesn't produce an identity. It is however "related" to the identity functor. The combination left-after-right produces an object that can be mapped back *to* the original object. Conversely, right-after-left has a mapping *from* the identity functor. These two give rise to natural transformations that are called, respectively, the counit ε and the unit η.



Here, the combination diagonal functor *after* the product functor takes a pair $\langle a, b\rangle$ to the pair $\langle a \times b, a \times b\rangle$. The counit ε then maps it back to $\langle a, b\rangle$ using a pair of projections $\langle \pi_1, \pi_2\rangle$ (which is a single morphism in $C \times C$). It's easy to see that the family of such morphisms defines a natural transformation.

If we think for a moment in terms of set elements, then for every element of the target object, the counit *extracts* a pair of elements of the source object (the objects here are pairs of sets). Note that this mapping is not injective and, therefore, not invertible.

The other composition–the product functor after the diagonal functor–maps an object c to $c \times c$. The component of the unit natural transformation, $\eta_c \colon c \to c \times c$, is implemented using the universal property of the product. Indeed, such a morphism is uniquely determined by a pair of identity morphsims $\langle id_c, id_c\rangle$. Again, when we vary c, these morphisms combine to form a natural transformation.

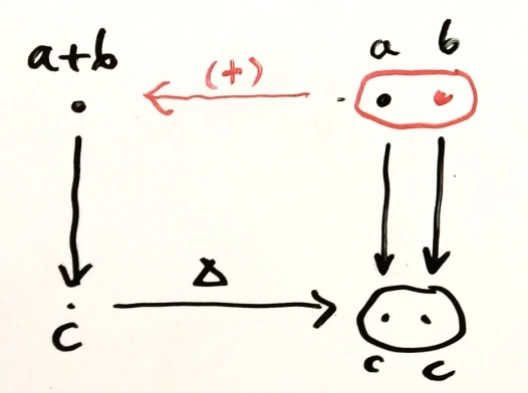
Thinking in terms of set elements, the unit *inserts* an element of the set c in the target set. And again, this is not an injective map, so it cannot be inverted.

Although in an arbitrary category we cannot talk about elements, a lot of intuitions from Set carry over to a more general setting. In a category with a terminal object, for instance, we can talk about *global elements* as mappings from the terminal object. In the absence of the terminal object, we may use other objects to define *generalized elements*. This is all in the true spirit of category theory, which defines all properties of objects in terms of morphisms.

Every construction in category theory has its dual, and the product is no exception.

A *coproduct* is defined by a mapping out property. For every pair of morphisms from, respectively, a and b to the common target c there is a unique mapping out from the coproduct $a + b$ to c. In programming, this is called *case analysis*: a function from a sum type is implemented using two functions corresponding to two cases. Conversely, given a mapping out of a coproduct, the two functions are uniquely determined due to the commuting conditions (this was all discussed in the previous post).

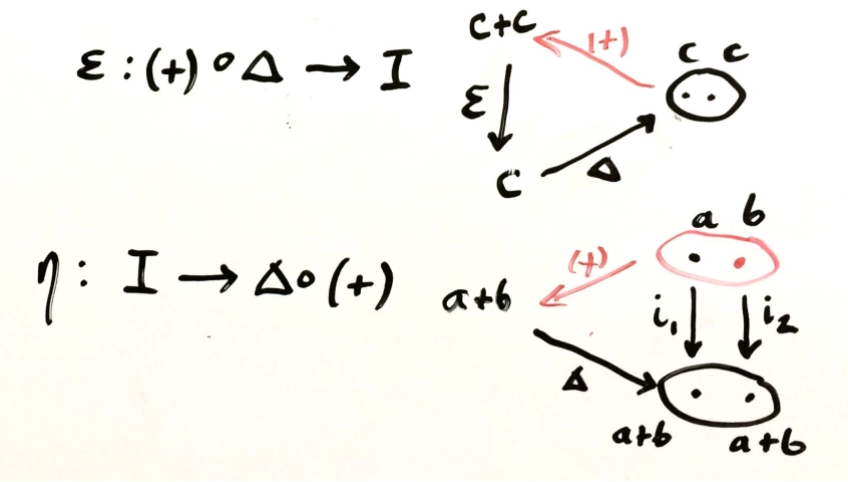
As before, this one-to-one correspondence can be neatly encapsulated as an adjunction. This time, however, the coproduct functor is the left adjoint of the diagonal functor.



The coproduct is still the "forgetful" part of the duo, but now the diagonal functor plays the role of the *cofree* funtor, relative to the coproduct. Again, I'm using these terms loosely.

The counit now works in the category C and it "extracts a value" from the symmetric coproduct of c with c. It does it by "pattern matching" and applying the identity morphism.

The unit is more interesting. It's built from two injections, or two constructors, as we call them in programming.



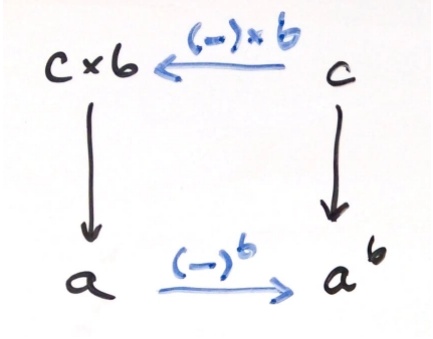
I find it fascinating that the simple diagonal functor can be used to define both products and coproducts. Moreover, using terse categorical notation, this whole blog post up to this point can be summarized by a single formula.

That's the power of adjunctions.

There is one more very important adjunction that every programmer should know: the exponential, or the currying adjunction. The exponential, a.k.a. the function type, is the right adjoint to the product functor. What's the product functor? Product is a bifunctor, or a functor from $C \times C$ to C. But if you fix one of the arguments, it just becomes a regular functor. We're interested in the functor $(-) \times b$ or, more explicitly:

$$(-) \times b : a \to a \times b$$

It's a functor that multiplies its argument by some fixed object b. We are using this functor to define the exponential. The exponential a^b is defined by the mapping-in property. The mappings out of the product $c \times b$ to a are in one to one correspondence with morphisms from an arbitrary object c to the exponential a^b.

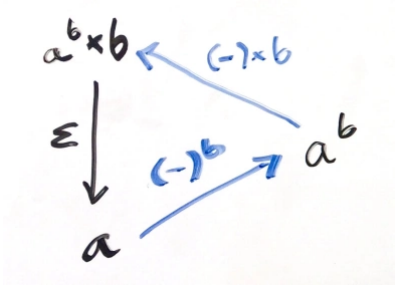


$$C(c \times b, a) \cong C(c, a^b)$$

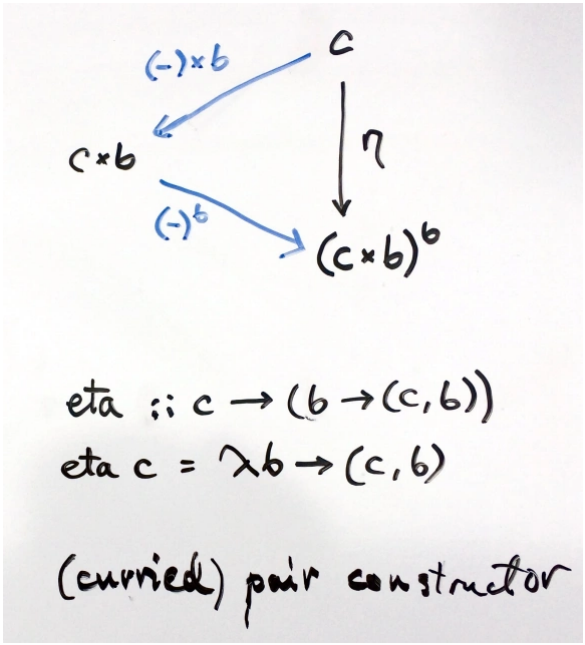
The exponential a^b is an object representing the set of morphisms from b to a, and the two directions of the isomorphism above are called *curry* and *uncurry*.

This is exactly the meaning of the universal property of the exponential I discussed in my previous post.

The counit for this adjunction extracts a value from the product of the function type (the exponential) and its argument. It's just the evaluation morphism: it applies a function to its argument.



The unit injects a value of type c into a function type $b \to c \times b$. The unit is just the curried version of the product constructor.



I want you to look closely at this formula through your programming glasses. The target of the unit is the type:

```
b -> (c, b)
```

You may recognize it as the state monad, with `b` representing the state. The unit is nothing else but the natural transformation whose component we call `return`. Coincidence? Not really! Look at the component of the counit:

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

It's the `extract` of the `Store` comonad.

It turns out that every adjunction gives rise to both a monad and a comonad. Not only that, every monad and every comonad give rise to adjunctions.

It seems like, in category theory, if you dig deep enough, everything is related to everything in some meaningful way. And every time you revisit a topic, you discover new insights. That's what makes category theory so exciting.

March 27, 2019

Promonads, Arrows, and Einstein Notation for Profunctors

Posted by Bartosz Milewski under Category Theory, Haskell, Monads, Programming
1 Comment

i
14 Votes

I've been working with profunctors lately. They are interesting beasts, both in category theory and in programming. In Haskell, they form the basis of profunctor optics–in particular the lens library.

Profunctor Recap

The categorical definition of a profunctor doesn't even begin to describe its richness. You might say that it's just a functor from a product category $\mathbb{C}^{op} \times \mathbb{D}$ to Set (I'll stick to Set for simplicity, but there are generalizations to other categories as well).

A profunctor P (a.k.a., a distributor, or bimodule) maps a pair of objects, c from $\mathbb{C}$ and d from $\mathbb{D}$, to a set $P(c, d)$. Being a functor, it also maps any pair of morphisms in $\mathbb{C}^{op} \times \mathbb{D}$:

$$f\colon c' \to c$$
$$g\colon d \to d'$$

to a function between those sets:

$$P(f, g)\colon P(c, d) \to P(c', d')$$

Notice that the first morphism f goes in the opposite direction to what we normally expect for functors. We say that the profunctor is *contravariant* in its first argument and *covariant* in the second.

But what's so special about this particular combination of source and target categories?

Hom-Profunctor

The key point is to realize that a profunctor generalizes the idea of a hom-functor. Like a profunctor, a hom-functor maps pairs of objects to sets. Indeed, for any two objects in $\mathbb{C}$ we have the set of morphisms between them, $C(a, b)$.

Also, any pair of morphisms in $\mathbb{C}$:

$$f\colon a' \to a$$
$$g\colon b \to b'$$

can be lifted to a function, which we will denote by $C(f, g)$, between hom-sets:

$$C(f, g) \colon C(a, b) \to C(a', b')$$

Indeed, for any $h \in C(a, b)$ we have:

$$C(f, g)h = g \circ h \circ f \in C(a', b')$$

This (plus functorial laws) completes the definition of a functor from $\mathbb{C}^{op} \times \mathbb{C}$ to Set. So a hom-functor is a special case of an endo-profunctor (where $\mathbb{D}$ is the same as $\mathbb{C}$). It's contravariant in the first argument and covariant in the second.

For Haskell programmers, here's the definition of a profunctor from Edward Kmett's `Data.Profunctor` library:

```
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'
```

The function `dimap` does the lifting of a pair of morphisms.

Here's the proof that the hom-functor which, in Haskell, is represented by the arrow `->`, is a profunctor:

```
instance Profunctor (->) where
  dimap ab cd bc = cd . bc . ab
```

Not only that: a general profunctor can be considered an extension of a hom-functor that forms a bridge between two categories. Consider a profunctor P spanning two categories $\mathbb{C}$ and $\mathbb{D}$:

$$P \colon \mathbb{C}^{op} \times \mathbb{D} \to Set$$

For any two objects from one of the categories we have a regular hom-set. But if we take one object c from $\mathbb{C}$ and another object d from $\mathbb{D}$, we can generate a set $P(c, d)$. This set works just like a hom-set. Its elements are called *heteromorphisms*, because they can be thought of as representing morphism between two different categories. What makes them similar to morphisms is that they can be composed with regular morphisms. Suppose you have a morphism in $\mathbb{C}$:

$$f \colon c' \to c$$

and a heteromorphism $h \in P(c, d)$. Their composition is another heteromorphism obtained by lifting the pair (f, id_d). Indeed:

$$P(f, id_d) \colon P(c, d) \to P(c', d)$$

so its action on h produces a heteromorphism from c' to d, which we can call the *composition* $h \circ f$ of a heteromorphism h with a morphism f. Similarly, a morphism in $\mathbb{D}$:

$$g \colon d \to d'$$

can be composed with h by lifting (id_c, g).

In Haskell, this new composition would be implemented by applying `dimap f id` to precompose `p c d` with

```
f :: c' -> c
```

and `dimap id g` to postcompose it with

```
g :: d -> d'
```

This is how we can use a profunctor to glue together two categories. Two categories connected by a profunctor form a new category known as their *collage*.

A given profunctor provides unidirectional flow of heteromorphisms from $\mathbb{C}$ to $\mathbb{D}$, so there is no opportunity to compose two heteromorphisms.

Profunctors As Relations

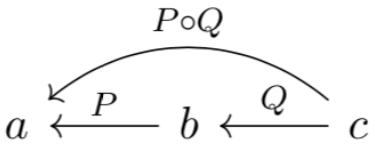
The opportunity to compose heteromorphisms arises when we decide to glue more than two categories. The clue as how to proceed comes from yet another interpretation of profunctors: as proof-relevant relations. In classical logic, a *relation* between sets assigns a Boolean *true* or *false* to each pair of elements. The elements are either related or not, period. In proof-relevant logic, we are not only interested in whether something is true, but also in gathering witnesses to the proofs. So, instead of assigning a single Boolean to each pair of elements, we assign a whole set. If the set is empty, the elements are unrelated. If it's non-empty, each element is a separate witness to the relation.

This definition of a relation can be generalized to any category. In fact there is already a natural relation between objects in a category–the one defined by hom-sets. Two objects a and b are related this way if the hom-set $C(a, b)$ is non-empty. Each morphism in $C(a, b)$ serves as a witness to this relation.

With profunctors, we can define proof-relevant relations between objects that are taken from different categories. Object c in $\mathbb{C}$ is related to object d in $\mathbb{D}$ if $P(c, d)$ is a non-empty set. Moreover, each element of this set serves as a witness for the relation. Because of functoriality of P, this relation is compatible with the categorical structure, that is, it composes nicely with the relation defined by hom-sets.

In general, a composition of two relations P and Q, denoted by $P \circ Q$ is defined as a path between objects. Objects a and c are related if there is a go-between object b such that both $P(a, b)$ and $Q(b, c)$ are non-empty. As a witness of this relation we can pick any pair of elements, one from $P(a, b)$ and one from $Q(b, c)$.

By convention, a profunctor $P(a, b)$ is drawn as an arrow (often crossed) from b to a, $a \nleftarrow b$.



Composition of profunctors / relations

Profunctor Composition

To create a set of all witnesses of $P \circ Q$ we have to sum over all possible intermediate objects and all pairs of witnesses. Roughly speaking, such a sum (modulo some identifications) is expressed categorically as a coend:

$$(P \circ Q)(a, c) = \int^b P(a, b) \times Q(b, c)$$

As a refresher, a coend of a profunctor P is a set $\int^a P(a, a)$ equipped with a family of injections

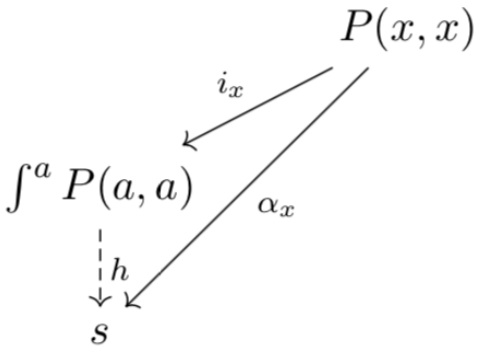
$$i_x \colon P(x, x) \to \int^a P(a, a)$$

that is universal in the sense that, for any other set s and a family:

$$\alpha_x \colon P(x, x) \to s$$

there is a unique function h that factorizes them all:

$$\alpha_x = h \circ i_x$$



Universal property of a coend

Profunctor composition can be translated into pseudo-Haskell as:

```
type Procompose q p a c = exists b. (p a b, q b c)
```

where the coend is encoded as an existential data type. The actual implementation (again, see Edward Kmett's `Data.Profunctor.Composition`) is:

```
data Procompose q p a c where
  Procompose :: q b c -> p a b -> Procompose q p a c
```

The existential quantifier is expressed in terms of a GADT (Generalized Algebraic Data Type), with the free occurrence of `b` inside the data constructor.

Einstein's Convention

By now you might be getting lost juggling the variances of objects appearing in those formulas. The coend variable, for instance, must appear under the integral sign once in the covariant and once in the contravariant position, and the variances on the right must match the variances on the left. Fortunately, there is a precedent in a different branch of mathematics, tensor calculus in vector spaces, with the kind of notation that takes care of variances. Einstein coopted and expanded this notation in his theory of relativity. Let's see if we can adapt this technique to the calculus of profunctors.

The trick is to write contravariant indices as superscripts and the covariant ones as subscripts. So, from now on, we'll write the components of a profunctor p (we'll switch to lower case to be compatible with Haskell) as $p^c{}_d$. Einstein also came up with a clever convention: implicit summation over a repeated index. In the case of profunctors, the summation corresponds to taking a coend. In this notation, a coend over a profunctor p looks like a trace of a tensor:

$$p^a{}_a = \int^a p(a, a)$$

The composition of two profunctors becomes:

$$(p \circ q)^a{}_c = p^a{}_b \, q^b{}_c = \int^b p(a, b) \times q(b, c)$$

The summation convention applies only to adjacent indices. When they are separated by an explicit product sign (or any other operator), the coend is not assumed, as in:

$$p^a{}_b \times q^b{}_c$$

(no summation).

The hom-functor in a category $\mathbb{C}$ is also a profunctor, so it can be notated appropriately:

$$C^a{}_b = C(a, b)$$

The co-Yoneda lemma (see Ninja Yoneda) becomes:

$$C^c{}_{c'} \, p^{c'}{}_d \cong p^c{}_d \cong p^c{}_{d'} \, D^{d'}{}_d$$

suggesting that the hom-functors $C^c{}_{c'}$ and $D^{d'}{}_d$ behave like Kronecker deltas (in tensor-speak) or unit matrices. Here, the profunctor p spans two categories

$$p \colon \mathbb{C}^{op} \times \mathbb{D} \to Set$$

The lifting of morphisms:

$$f : c' \to c$$
$$g : d \to d'$$

can be written as:

$$p^f{}_g : p^c{}_d \to p^{c'}{}_{d'}$$

There is one more useful identity that deals with mapping out from a coend. It's the consequence of the fact that the hom-functor is continuous. It means that it maps (co-) limits to limits. More precisely, since the hom-functor is contravariant in the first variable, when we fix the target object, it maps colimits in the first variable to limits. (It also maps limits to limits in the second variable). Since a coend is a colimit, and an end is a limit, continuity leads to the following identity:

$$Set(\int^c p(c,c), s) \cong \int_c Set(p(c,c), s)$$

for any set s. Programmers know this identity as a generalization of case analysis: a function from a sum type is a product of functions (one function per case). If we interpret the coend as an existential quantifier, the end is equivalent to a universal quantifier.

Let's apply this identity to the mapping out from a composition of two profunctors:

$$p^a{}_b\, q^b{}_c \to s = Set\Big(\int^b p(a,b) \times q(b,c), s\Big)$$

This is isomorphic to:

$$\int_b Set\Big(p(a,b) \times q(b,c), s\Big)$$

or, after currying (using the product/exponential adjunction),

$$\int_b Set\Big(p(a,b), q(b,c) \to s\Big)$$

This gives us the mapping out formula:

$$p^a{}_b\, q^b{}_c \to s \cong p^a{}_b \to q^b{}_c \to s$$

with the right hand side natural in b. Again, we don't perform implicit summation on the right, where the repeated indices are separated by an arrow. There, the repeated index b is universally quantified (through the end), giving rise to a natural transformation.

Bicategory Prof

Since profunctors can be composed using the coend formula, it's natural to ask if there is a category in which they work as morphisms. The only problem is that profunctor composition satisfies the associativity and unit laws (see the co-Yoneda lemma above) only up to isomorphism. Not to worry, there is a name for that: a *bicategory*. In a bicategory we have objects, which are called 0-cells; morphisms,

which are called 1-cells; and morphisms between morphisms, which are called 2-cells. When we say that categorical laws are satisfied up to isomorphism, it means that there is an invertible 2-cell that maps one side of the law to another.

The bicategory $Prof$ has categories as 0-cells, profunctors as 1-cells, and natural transformations as 2-cells. A natural transformation α between profunctors p and q

$$\alpha \colon p \Rightarrow q$$

has components that are functions:

$$\alpha^c{}_d \colon p^c{}_d \to q^c{}_d$$

satisfying the usual naturality conditions. Natural transformations between profunctors can be composed as functions (this is called vertical composition). In fact 2-cells in any bicategory are composable, and there always is a unit 2-cell. It follows that 1-cells between any two 0-cells form a category called the hom-category.

But there is another way of composing 2-cells that's called horizontal composition. In $Prof$, this horizontal composition is not the usual horizontal composition of natural transformations, because composition of profunctors is not the usual composition of functors. We have to construct a natural transformation between one composition of profuntors, say $p^a{}_b \, q^b{}_c$ and another, $r^a{}_b \, s^b{}_c$, having at our disposal two natural transformations:

$$\alpha \colon p \Rightarrow r$$

$$\beta \colon q \Rightarrow s$$

The construction is a little technical, so I'm moving it to the appendix. We will denote such horizontal composition as:

$$(\alpha \circ \beta)^a{}_c \colon p^a{}_b \, q^b{}_c \to r^a{}_b \, s^b{}_c$$

If one of the natural transformations is an identity natural transformation, say, from $p^a{}_b$ to $p^a{}_b$, horizontal composition is called *whiskering* and can be written as:

$$(p \circ \beta)^a{}_c \colon p^a{}_b \, q^b{}_c \to p^a{}_b \, s^b{}_c$$

Promonads

The fact that a monad is a monoid in the category of endofunctors is a lucky accident. That's because, in general, a monad can be defined in any bicategory, and Cat just happens to be a (strict) bicategory. It has (small) categories as 0-cells, functors as 1-cells, and natural transformations as 2-cells. A monad is defined as a combination of a 0-cell (you need a category to define a monad), an endo-1-cell (that would be an endofunctor in that category), and two 2-cells. These 2-cells are variably called multiplication and unit, μ and η, or `join` and `return`.

Since $Prof$ is a bicategory, we can define a monad in it, and call it a promonad. A promonad consists of a 0-cell C, which is a category; an endo-1-cell p, which is a profunctor in that category; and two 2-cells, which are natural transformations:

$$\mu^a{}_b \colon p^a{}_c \, p^c{}_b \to p^a{}_b$$

$$\eta^a{}_b \colon C^a{}_b \to p^a{}_b$$

Remember that $C^a{}_b$ is the hom-profunctor in the category C which, due to co-Yoneda, happens to be the unit of profunctor composition.

Programmers might recognize elements of the Haskell `Arrow` in it (see my blog post on monoids).

We can apply the mapping-out identity to the definition of multiplication and get:

$$\mu^a{}_b \colon p^a{}_c \to p^c{}_b \to p^a{}_b$$

Notice that this looks very much like composition of heteromorphisms. Moreover, the monadic unit η maps regular morphisms to heteromorphisms. We can then construct a new category, whose objects are the same as the objects of $\mathbb{C}$, with hom-sets given by the profunctor p. That is, a hom set from a to b is the set $p^a{}_b$. We can define an identity-on-object functor J from $\mathbb{C}$ to that category, whose action on hom-sets is given by η.

Interestingly, this construction also works in the opposite direction (as was brought to my attention by Alex Campbell). Any indentity-on-objects functor defines a promonad. Indeed, given a functor J, we can always turn it into a profunctor:

$$p(c, d) = D(J\, c, J\, d)$$

In Einstein notation, this reads:

$$p^c{}_d = D^{J\, c}{}_{J\, d}$$

Since J is identity on objects, the composition of morphisms in D can be used to define the composition of heteromorphisms. This, in turn, can be used to define μ, thus showing that p is a promonad on $\mathbb{C}$.

Conclusion

I realize that I have touched upon some pretty advanced topics in category theory, like bicategories and promonads, so it's a little surprising that these concepts can be illustrated in Haskell, some of them being present in popular libraries, like the `Arrow` library, which has applications in functional reactive programming.

I've been experimenting with applying Einstein's summation convention to profunctors, admittedly with mixed results. This is definitely work in progress and I welcome suggestions to improve it. The main problem is that we sometimes need to apply the sum (coend), and at other times the product (end)

to repeated indices. This is in particular awkward in the formulation of the mapping out property. I suggest separating the non-summed indices with product signs or arrows but I'm not sure how well this will work.

Appendix: Horizontal Composition in Prof

We have at our disposal two natural transformations:

$\alpha \colon p \Rightarrow r$

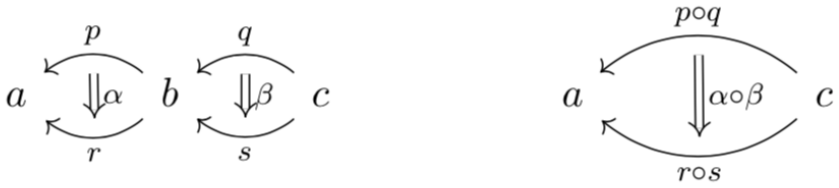
$\beta \colon q \Rightarrow s$

and the following coend, which is the composition of the profunctors p and q:

$\int^b p(a,b) \times q(b,c)$

Our goal is to construct an element of the target coend:

$\int^b r(a,b) \times s(b,c)$



Horizontal composition of 2-cells

To construct an element of a coend, we need to provide just one element of $r(a,b') \times s(b',c)$ for some b'. We'll look for a function that would construct such an element in the following hom-set:

$Set\Big(\int^b p(a,b) \times q(b,c), r(a,b') \times s(b',c) \Big)$

Using Einstein notation, we can write it as:

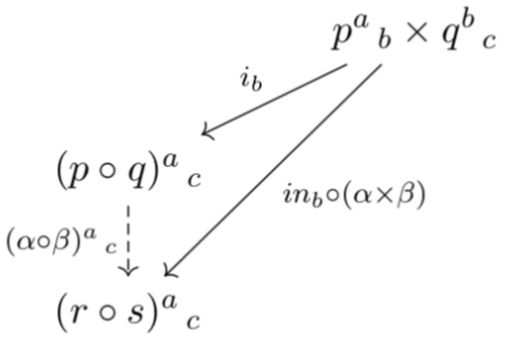
$p^a{}_b \, q^b{}_c \to r^a{}_{b'} \times s^{b'}{}_c$

and then use the mapping out property:

$p^a{}_b \to q^b{}_c \to r^a{}_{b'} \times s^{b'}{}_c$

We can pick b' equal to b and implement the function using the components of the two natural transformations, $\alpha^a{}_b \times \beta^b{}_c$.

Of course, this is how a programmer might think of it. A mathematician will use the universal property of the coend $(p \circ q)^a{}_c$, as in the diagram below (courtesy Alex Campbell).



Horizontal composition using the universal property of a coend

In Haskell, we can define a natural transformation between two (endo-) profunctors as a polymorphic function:

```
newtype PNat p q = PNat (forall a b. p a b -> q a b)
```

Horizontal composition is then given by:

```
horPNat :: PNat p r -> PNat q s -> Procompose p q a c
        -> Procompose r s a c
horPNat (PNat alpha) (PNat beta) (Procompose pbc qdb) =
  Procompose (alpha pbc) (beta qdb)
```

Acknowledgment

I'm grateful to Alex Campbell from Macquarie University in Sydney for extensive help with this blog post.

Further Reading

- Dominique Bourn et Jacques Penon, 2-Catégories Réductibles
- Ross Street, Cauchy characterization of enriched categories

January 5, 2019

ACT2019 School: Call for Participation

Posted by Bartosz Milewski under Category Theory
Leave a Comment

i
6 Votes

Oxford, UK. 2019 July 22 – 26

Dear scientists, mathematicians, linguists, philosophers, and hackers,

We are writing to let you know about a fantastic opportunity to learn about the emerging interdisciplinary field of applied category theory from some of its leading researchers at the ACT2019 School. It will begin in January 2019 and culminate in a meeting in Oxford, July 22-26.

Applied category theory is a topic of interest for a growing community of researchers, interested in studying systems of all sorts using category-theoretic tools. These systems are found in the natural sciences and social sciences, as well as in computer science, linguistics, and engineering. The background and experience of our community's members are as varied as the systems being studied.

The goal of the ACT2019 School is to help grow this community by pairing ambitious young researchers together with established researchers in order to work on questions, problems, and conjectures in applied category theory.

Who should apply?

Anyone from anywhere who is interested in applying category-theoretic methods to problems outside of pure mathematics. This is emphatically not restricted to math students, but one should be comfortable working with mathematics. Knowledge of basic category-theoretic language—the definition of monoidal category for example— is encouraged.

We will consider advanced undergraduates, Ph.D. students, and post-docs. We ask that you commit to the full program as laid out below.

Instructions on how to apply can be found below the research topic descriptions.

Senior research mentors and their topics

Below is a list of the senior researchers, each of whom describes a research project that their team will pursue, as well as the background reading that will be studied between now and July 2019.

Miriam Backens

Title: *Simplifying quantum circuits using the ZX-calculus*

Description: The ZX-calculus is a graphical calculus based on the category-theoretical formulation of quantum mechanics. A complete set of graphical rewrite rules is known for the ZX-calculus, but not for quantum circuits over any universal gate set. In this project, we aim to develop new strategies for using the ZX-calculus to simplify quantum circuits.

Background reading:

1. Matthes Amy, Jianxin Chen, Neil Ross. A finite presentation of CNOT-Dihedral operators. arXiv:1701.00140
2. Miriam Backens. The ZX-calculus is complete for stabiliser quantum mechanics. arXiv:1307.7025

Tobias Fritz

Title: *Partial evaluations, the bar construction, and second-order stochastic dominance*

Description: We all know that 2+2+1+1 evaluates to 6. A less familiar notion is that it can *partially evaluate* to 5+1. In this project, we aim to study the compositional structure of partial evaluation in terms of monads and the bar construction and see what this has to do with financial risk via second-order stochastic dominance.

Background reading:

1. Tobias Fritz, Paolo Perrone. Monads, partial evaluations, and rewriting. arXiv:1810.06037
2. Maria Manuel Clementino, Dirk Hofmann, George Janelidze. The monads of classical algebra are seldom weakly cartesian. Available here.
3. Todd Trimble. On the bar construction. Available here.

Pieter Hofstra

Title: *Complexity classes, computation, and Turing categories*

Description: Turing categories form a categorical setting for studying computability without bias towards any particular model of computation. It is not currently clear, however, that Turing categories are useful to study practical aspects of computation such as complexity. This project revolves around the systematic study of step-based computation in the form of stack-machines, the resulting Turing categories, and complexity classes. This will involve a study of the interplay between traced monoidal structure and computation. We will explore the idea of stack machines qua programming languages, investigate the expressive power, and tie this to complexity theory. We will also consider questions such as the following: can we characterize Turing categories arising from stack machines? Is there an initial such category? How does this structure relate to other categorical structures associated with computability?

Background reading:

1. J.R.B. Cockett, P.J.W. Hofstra. Introduction to Turing categories. APAL, Vol 156, pp 183-209, 2008. Available here .
2. J.R.B. Cockett, P.J.W. Hofstra, P. Hrubes. Total maps of Turing categories. ENTCS (Proc. of MFPS XXX), pp 129-146, 2014. Available here.
3. A. Joyal, R. Street, D. Verity. Traced monoidal categories. Mat. Proc. Cam. Phil. Soc. 3, pp. 447-468, 1996. Available here.

Bartosz Milewski

Title: *Traversal optics and profunctors*

Description: In functional programming, optics are ways to zoom into a specific part of a given data type and mutate it. Optics come in many flavors such as lenses and prisms and there is a well-studied categorical viewpoint, known as profunctor optics. Of all the optic types, only the traversal has resisted a derivation from first principles into a profunctor description. This project aims to do just this.

Background reading:

1. Bartosz Milewski. Profunctor optics, categorical View. Available here.
2. Craig Pastro, Ross Street. Doubles for monoidal categories. arXiv:0711.1859

Mehrnoosh Sadrzadeh

Title: *Formal and experimental methods to reason about dialogue and discourse using categorical models of vector spaces*

Description: Distributional semantics argues that meanings of words can be represented by the frequency of their co-occurrences in context. A model extending distributional semantics from words to sentences has a categorical interpretation via Lambek's syntactic calculus or pregroups. In this project, we intend to further extend this model to reason about dialogue and discourse utterances where people interrupt each other, there are references that need to be resolved, disfluencies, pauses, and corrections. Additionally, we would like to design experiments and run toy models to verify predictions of the developed models.

Background reading:

1. Gerhard Jager. A multi-modal analysis of anaphora and ellipsis. Available here.
2. Matthew Purver, Ronnie Cann, Ruth Kempson. Grammars as parsers: Meeting the dialogue challenge. Available here.

David Spivak

Title: *Toward a mathematical foundation for autopoiesis*

Description: An autopoietic organization—anything from a living animal to a political party to a football team—is a system that is responsible for adapting and changing itself, so as to persist as events unfold. We want to develop mathematical abstractions that are suitable to found a scientific study of autopoietic organizations. To do this, we'll begin by using behavioral mereology and graphical logic to frame a discussion of autopoiesis, most of all what it is and how it can be best conceived. We do not expect to complete this ambitious objective; we hope only to make progress toward it.

Background reading:

1. Fong, Myers, Spivak. Behavioral mereology. arXiv:1811.00420.
2. Fong, Spivak. Graphical regular logic. arXiv:1812.05765.
3. Luhmann. Organization and Decision, CUP. (Preface)

School structure

All of the participants will be divided up into groups corresponding to the projects. A group will consist of several students, a senior researcher, and a TA. Between January and June, we will have a reading course devoted to building the background necessary to meaningfully participate in the projects. Specifically, two weeks are devoted to each paper from the reading list. During this two week period, everybody will read the paper and contribute to a discussion in a private online chat forum. There will

be a TA serving as a domain expert and moderating this discussion. In the middle of the two week period, the group corresponding to the paper will give a presentation via video conference. At the end of the two week period, this group will compose a blog entry on this background reading that will be posted to the n-category cafe.

After all of the papers have been presented, there will be a two-week visit to Oxford University from 15 – 26 July 2019. The first week is solely for participants of the ACT2019 School. Groups will work together on research projects, led by the senior researchers.

The second week of this visit is the ACT2019 Conference, where the wider applied category theory community will arrive to share new ideas and results. It is not part of the school, but there is a great deal of overlap and participation is very much encouraged. The school should prepare students to be able to follow the conference presentations to a reasonable degree.

How to apply

To apply please send the following to **act2019school@gmail.com**

- Your CV
- A document with:
  - An explanation of any relevant background you have in category theory or any of the specific projects areas
  - The date you completed or expect to complete your Ph.D. and a one-sentence summary of its subject matter.
  - Order of project preference
  - To what extent can you commit to coming to Oxford (availability of funding is uncertain at this time)
- A brief statement (~300 words) on why you are interested in the ACT2019 School. Some prompts:
  - how can this school contribute to your research goals
  - how can this school help in your career?

Also, have sent on your behalf to act2019school@gmail.com a brief letter of recommendation confirming any of the following:

- your background
- ACT2019 School's relevance to your research/career
- your research experience

Questions?

For more information, contact either

- Daniel Cicala. cicala (at) math (dot) ucr (dot) edu
- Jules Hedges. julian (dot) hedges (at) cs (dot) ox (dot) ac (dot) uk

December 20, 2018

Open Season on Hylomorphisms

Posted by Bartosz Milewski under Category Theory, Functional Programming, Haskell, Programming [15] Comments

i
4 Votes

Yes, it's this time of the year again! I started a little tradition a year ago with Stalking a Hylomorphism in the Wild. This year I was reminded of the Advent of Code by a tweet with this succint C++ program:

```
int main() {
    auto hamming_distance = [](auto&& r1, auto&& r2) {
        return accumulate(view::zip(r1, r2), 0, ranges::plus{},
            [](auto&& x) { return x.first != x.second; });
    };

    auto ns = ranges::istream_range<std::string>(std::cin) | to_vector;
    auto found = view::cartesian_product(ns, ns)
               | view::filter([&](auto&& p) {
                   return hamming_distance(get<0>(p), get<1>(p)) == 1;
                 });
    for (auto [s1,s2] : found | view::take(1)) {
        for (auto[c1, c2] : view::zip(s1, s2)) {
            if (c1 == c2) std::cout << c1;
        }
    }
}
```

This piece of code is probably unreadable to a regular C++ programmer, but makes perfect sense to a Haskell programmer.

Here's the description of the problem: You are given a list of equal-length strings. Every string is different, but two of these strings differ only by one character. Find these two strings and return their matching part. For instance, if the two strings were "abcd" and "abxd", you would return "abd".

What makes this problem particularly interesting is its potential application to a much more practical task of matching strands of DNA while looking for mutations. I decided to explore the problem a little beyond the brute force approach. And, of course, I had a hunch that I might encounter my favorite wild beast–the hylomorphism.

Brute force approach

First things first. Let's do the boring stuff: read the file and split it into lines, which are the strings we are supposed to process. So here it is:

```
main = do
  txt <- readFile "day2.txt"
  let cs = lines txt
  print $ findMatch cs
```

The real work is done by the function `findMatch`, which takes a list of strings and produces the answer, which is a single string.

```
findMatch :: [String] -> String
```

First, let's define a function that calculates the distance between any two strings.

```
distance :: (String, String) -> Int
```

We'll define the distance as the count of mismatched characters.

Here's the idea: We have to compare strings (which, let me remind you, are of equal length) character by character. Strings are lists of characters. The first step is to take two strings and zip them together, producing a list of pairs of characters. In fact we can combine the zipping with the next operation–in this case, comparison for inequality, `(/=)`–using the library function `zipWith`. However, `zipWith` is defined to act on two lists, and we will want it to act on a pair of lists–a subtle distinction, which can be easily overcome by applying `uncurry`:

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

which turns a function of two arguments into a function that takes a pair. Here's how we use it:

```
uncurry (zipWith (/=))
```

The comparison operator `(/=)` produces a Boolean result, `True` or `False`. We want to count the number of differences, so we'll covert `True` to one, and `False` to zero:

```
fromBool :: Num a => Bool -> a
fromBool False = 0
fromBool True  = 1
```

(Notice that such subtleties as the difference between `Bool` and `Int` are blisfully ignored in C++.)

Finally, we'll sum all the ones using `sum`. Altogether we have:

```
distance = sum . fmap fromBool . uncurry (zipWith (/=))
```

Now that we know how to find the distance between any two strings, we'll just apply it to all possible pairs of strings. To generate all pairs, we'll use list comprehension:

```
let ps = [(s1, s2) | s1 <- ss, s2 <- ss]
```

(In C++ code, this was done by `cartesian_product`.)

Our goal is to find the pair whose distance is exactly one. To this end, we'll apply the appropriate filter:

```
filter ((== 1) . distance) ps
```

For our purposes, we'll assume that there is exactly one such pair (if there isn't one, we are willing to let the program fail with a fatal exception).

```
(s, s') = head $ filter ((== 1) . distance) ps
```

The final step is to remove the mismatched character:

```
filter (uncurry (==)) $ zip s s'
```

We use our friend `uncurry` again, because the equality operator `(==)` expects two arguments, and we are calling it with a pair of arguments. The result of filtering is a list of identical pairs. We'll `fmap fst` to pick the first components.

```
findMatch :: [String] -> String
findMatch ss =
  let ps = [(s1, s2) | s1 <- ss, s2 <- ss]
      (s, s') = head $ filter ((== 1) . distance) ps
  in fmap fst $ filter (uncurry (==)) $ zip s s'
```

This program produces the correct result and we could stop right here. But that wouldn't be much fun, would it? Besides, it's possible that other algorithms could perform better, or be more flexible when applied to a more general problem.

Data-driven approach

The main problem with our brute-force approach is that we are comparing everything with everything. As we increase the number of input strings, the number of comparisons grows like a factorial. There is a standard way of cutting down on the number of comparison: organizing the input into a neat data structure.

We are comparing strings, which are lists of characters, and list comparison is done recursively. Assume that you know that two strings share a prefix. Compare the next character. If it's equal in both strings, recurse. If it's not, we have a single character fault. The rest of the two strings must now match perfectly to be considered a solution. So the best data structure for this kind of algorithm should batch together strings with equal prefixes. Such a data structure is called a *prefix tree*, or a *trie* (pronounced *try*).

At every level of our prefix tree we'll branch based on the current character (so the maximum branching factor is, in our case, 26). We'll record the character, the count of strings that share the prefix that led us there, and the child trie storing all the suffixes.

```
data Trie = Trie [(Char, Int, Trie)]
  deriving (Show, Eq)
```

Here's an example of a trie that stores just two strings, "abcd" and "abxd". It branches after `b`.

```
   a 2
   b 2
c 1    x 1
d 1    d 1
```

When inserting a string into a trie, we recurse both on the characters of the string and the list of branches. When we find a branch with the matching character, we increment its count and insert the rest of the string into its child trie. If we run out of branches, we create a new one based on the current character, give it the count one, and the child trie with the rest of the string:

```
insertS :: Trie -> String -> Trie
insertS t "" = t
insertS (Trie bs) s = Trie (inS bs s)
  where
    inS ((x, n, t) : bs) (c : cs) =
      if c == x
      then (c, n + 1, insertS t cs) : bs
      else (x, n, t) : inS bs (c : cs)
    inS [] (c : cs) = [(c, 1, insertS (Trie []) cs)]
```

We convert our input to a trie by inserting all the strings into an (initially empty) trie:

```
mkTrie :: [String] -> Trie
mkTrie = foldl insertS (Trie [])
```

Of course, there are many optimizations we could use, if we were to run this algorithm on big data. For instance, we could compress the branches as is done in radix trees, or we could sort the branches alphabetically. I won't do it here.

I won't pretend that this implementation is simple and elegant. And it will get even worse before it gets better. The problem is that we are dealing explicitly with recursion in multiple dimensions. We recurse over the input string, the list of branches at each node, as well as the child trie. That's a lot of recursion to keep track of–all at once.

Now brace yourself: We have to traverse the trie starting from the root. At every branch we check the prefix count: if it's greater than one, we have more than one string going down, so we recurse into the child trie. But there is also another possibility: we can allow to have a mismatch at the current level. The current characters may be different but, since we allow only one mismatch, the rest of the strings have to match exactly. That's what the function `exact` does. Notice that `exact t` is used inside `foldMap`, which is a version of `fold` that works on monoids–here, on strings.

```
match1 :: Trie -> [String]
match1 (Trie bs) = go bs
  where
    go :: [(Char, Int, Trie)] -> [String]
    go ((x, n, t) : bs) =
      let a1s = if n > 1
                then fmap (x:) $ match1 t
                else []
          a2s = foldMap (exact t) bs
          a3s = go bs -- recurse over list
      in a1s ++ a2s ++ a3s
    go [] = []
    exact t (_, _, t') = matchAll t t'
```

Here's the function that finds all exact matches between two tries. It does it by generating all pairs of branches in which top characters match, and then recursing down.

```
matchAll :: Trie -> Trie -> [String]
matchAll (Trie bs) (Trie bs') = mAll bs bs'
  where
    mAll :: [(Char, Int, Trie)] -> [(Char, Int, Trie)] -> [String]
    mAll [] [] = [""]
    mAll bs bs' =
      let ps = [ (c, t, t')
               | (c,  _,  t)  <- bs
               , (c', _', t') <- bs'
               , c == c']
      in foldMap go ps
    go (c, t, t') = fmap (c:) (matchAll t t')
```

When `mAll` reaches the leaves of the trie, it returns a singleton list containing an empty string. Subsequent actions of `fmap (c:)` will prepend characters to this string.

Since we are expecting exactly one solution to the problem, we'll extract it using `head`:

```
findMatch1 :: [String] -> String
findMatch1 cs = head $ match1 (mkTrie cs)
```

Recursion schemes

As you hone your functional programming skills, you realize that explicit recursion is to be avoided at all cost. There is a small number of recursive patterns that have been codified, and they can be used to solve the majority of recursion problems (for some categorical background, see F-Algebras). Recursion itself can be expressed in Haskell as a data structure: a fixed point of a functor:

```
newtype Fix f = In { out :: f (Fix f) }
```

In particular, our trie can be generated from the following functor:

```
data TrieF a = TrieF [(Char, a)]
  deriving (Show, Functor)
```

Notice how I have replaced the recursive call to the `Trie` type constructor with the free type variable `a`. The functor in question defines the structure of a single node, leaving holes marked by the occurrences of `a` for the recursion. When these holes are filled with full blown tries, as in the definition of the fixed point, we recover the complete trie.

I have also made one more simplification by getting rid of the `Int` in every node. This is because, in the recursion scheme I'm going to use, the folding of the trie proceeds bottom-up, rather than top-down, so the multiplicity information can be passed upwards.

The main advantage of recursion schemes is that they let us use simpler, non-recursive building blocks such as algebras and coalgebras. Let's start with a simple coalgebra that lets us build a trie from a list of strings. A coalgebra is a fancy name for a particular type of function:

```
type Coalgebra f x = x -> f x
```

Think of `x` as a type for a seed from which one can grow a tree. A colagebra tells us how to use this seed to create a single node described by the functor `f` and populate it with (presumably smaller) seeds. We can then pass this coalgebra to a simple algorithm, which will recursively expand the seeds. This algorithm is called the *anamorphism*:

```
ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coa = In . fmap (ana coa) . coa
```

Let's see how we can apply it to the task of building a trie. The seed in our case is a list of strings (as per the definition of our problem, we'll assume they are all equal length). We start by grouping these strings into bunches of strings that start with the same character. There is a library function called `groupWith` that does exactly that. We have to import the right library:

```
import GHC.Exts (groupWith)
```

This is the signature of the function:

```
groupWith :: Ord b => (a -> b) -> [a] -> [[a]]
```

It takes a function `a -> b` that converts each list element to a type that supports comparison (as per the typeclass `Ord`), and partitions the input into lists that compare equal under this particular ordering. In our case, we are going to extract the first character from a string using `head` and bunch together all strings that share that first character.

```
let sss = groupWith head ss
```

The tails of those strings will serve as seeds for the next tier of the trie. Eventually the strings will be shortened to nothing, triggering the end of recursion.

```
fromList :: Coalgebra TrieF [String]
fromList ss =
  -- are strings empty? (checking one is enough)
  if null (head ss)
  then TrieF [] -- leaf
  else
    let sss = groupWith head ss
    in TrieF $ fmap mkBranch sss
```

The function `mkBranch` takes a bunch of strings sharing the same first character and creates a branch seeded with the suffixes of those strings.

```
mkBranch :: [String] -> (Char, [String])
mkBranch sss =
  let c = head (head sss) -- they're all the same
  in (c, fmap tail sss)
```

Notice that we have completely avoided explicit recursion.

The next step is a little harder. We have to fold the trie. Again, all we have to define is a step that folds a single node whose children have already been folded. This step is defined by an algebra:

```
type Algebra f x = f x -> x
```

Just as the type `x` described the seed in a coalgebra, here it describes the accumulator–the result of the folding of a recursive data structure.

We pass this algebra to a special algorithm called a *catamorphism* that takes care of the recursion:

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

Notice that the folding proceeds from the bottom up: the algebra assumes that all the children have already been folded.

The hardest part of designing an algebra is figuring out what information needs to be passed up in the accumulator. We obviously need to return the final result which, in our case, is the list of strings with one mismatched character. But when we are in the middle of a trie, we have to keep in mind that the mismatch may still happen above us. So we also need a list of strings that may serve as suffixes when the mismatch occurs. We have to keep them all, because they might be matched later with strings from other branches.

In other words, we need to be accumulating two lists of strings. The first list accumulates all suffixes for future matching, the second accumulates the results: strings with one mismatch (after the mismatch has been removed). We therefore should implement the following algebra:

```
Algebra TrieF ([String], [String])
```

To understand the implementation of this algebra, consider a single node in a trie. It's a list of branches, or pairs, whose first component is the current character, and the second a pair of lists of strings–the result of folding a child trie. The first list contains all the suffixes gathered from lower levels of the trie. The second list contains partial results: strings that were matched modulo single-character defect.

As an example, suppose that you have a node with two branches:

```
[ ('a', (["bcd", "efg"], ["pq"]))
, ('x', (["bcd"],        []))]
```

First we prepend the current character to strings in both lists using the function `prep` with the following signature:

```
prep :: (Char, ([String], [String])) -> ([String], [String])
```

This way we convert each branch to a pair of lists.

```
[ (["abcd", "aefg"], ["apq"])
, (["xbcd"],         [])]
```

We then merge all the lists of suffixes and, separately, all the lists of partial results, across all branches. In the example above, we concatenate the lists in the two columns.

```
(["abcd", "aefg", "xbcd"], ["apq"])
```

Now we have to construct new partial results. To do this, we create another list of accumulated strings from all branches (this time without prefixing them):

```
ss = concat $ fmap (fst . snd) bs
```

In our case, this would be the list:

```
["bcd", "efg", "bcd"]
```

To detect duplicate strings, we'll insert them into a multiset, which we'll implement as a map. We need to import the appropriate library:

```
import qualified Data.Map as M
```

and define a multiset `Counts` as:

```
type Counts a = M.Map a Int
```

Every time we add a new item, we increment the count:

```
add :: Ord a => Counts a -> a -> Counts a
add cs c = M.insertWith (+) c 1 cs
```

To insert all strings from a list, we use a fold:

```
mset = foldl add M.empty ss
```

We are only interested in items that have multiplicity greater than one. We can filter them and extract their keys:

```
dups = M.keys $ M.filter (> 1) mset
```

Here's the complete algebra:

```
accum :: Algebra TrieF ([String], [String])
accum (TrieF []) = ([""], [])
accum (TrieF bs) = -- b :: (Char, ([String], [String]))
    let -- prepend chars to string in both lists
        pss = unzip $ fmap prep bs
        (ss1, ss2) = both concat pss
        -- find duplicates
        ss = concat $ fmap (fst . snd) bs
        mset = foldl add M.empty ss
        dups = M.keys $ M.filter (> 1) mset
     in (ss1, dups ++ ss2)
  where
      prep :: (Char, ([String], [String])) -> ([String], [String])
      prep (c, pss) = both (fmap (c:)) pss
```

I used a handy helper function that applies a function to both components of a pair:

```
both :: (a -> b) -> (a, a) -> (b, b)
both f (x, y) = (f x, f y)
```

And now for the grand finale: Since we create the trie using an anamorphism only to immediately fold it using a catamorphism, why don't we cut the middle person? Indeed, there is an algorithm called the *hylomorphism* that does just that. It takes the algebra, the coalgebra, and the seed, and returns the fully charged accumulator.

```
hylo :: Functor f => Algebra f a -> Coalgebra f b -> b -> a
hylo alg coa = alg . fmap (hylo alg coa) . coa
```

And this is how we extract and print the final result:

```
print $ head $ snd $ hylo accum fromList cs
```

Conclusion

The advantage of using the hylomorphism is that, because of Haskell's laziness, the trie is never wholly constructed, and therefore doesn't require large amounts of memory. At every step enough of the data structure is created as is needed for immediate computation; then it is promptly released. In fact, the definition of the data structure is only there to guide the steps of the algorithm. We use a data structure as a control structure. Since data structures are much easier to visualize and debug than control structures, it's almost always advantageous to use them to drive computation.

In fact, you may notice that, in the very last step of the computation, our accumulator recreates the original list of strings (actually, because of laziness, they are never fully reconstructed, but that's not the point). In reality, the characters in the strings are never copied–the whole algorithm is just a choreographed dance of internal pointers, or iterators. But that's exactly what happens in the original C++ algorithm. We just use a higher level of abstraction to describe this dance.

I haven't looked at the performance of various implementations. Feel free to test it and report the results. The code is available on github.

Acknowledgments

I'm grateful to the participants of the Seattle Haskell Users' Group for many helpful comments during my presentation.

December 11, 2018

Keep it Simplex, Stupid!

Posted by Bartosz Milewski under Category Theory
[11] Comments

i
7 Votes

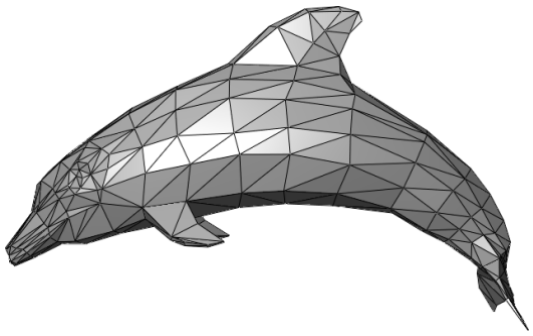
I wanted to do category theory, not geometry, so the idea of studying simplexes didn't seem very attractive at first. But as I was getting deeper into it, a very different picture emerged. Granted, the study of simplexes originated in geometry, but then category theorists took interest in it and turned it into something completely different. The idea is that simplexes define a very peculiar scheme for composing things. The way you compose lower dimensional simplexes in order to build higher dimensional simplexes forms a pattern that shows up in totally unrelated areas of mathematics… and programming. Recently I had a discussion with Edward Kmett in which he hinted at the simplicial structure of cumulative edits in a source file.

Geometric picture

Let's start with a simple idea, and see what we can do with it. The idea is that of triangulation, and it almost goes back to the beginning of the Agricultural Era. Somebody smart noticed long time ago that we can measure plots of land by subdividing them into triangles.

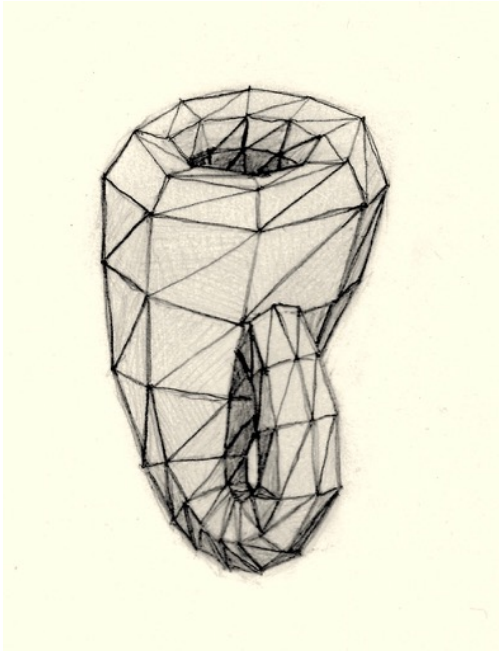
Why triangles and not, say, rectangles or quadrilaterals? Well, to begin with, a quadrilateral can be always divided into triangles, so triangles are more fundamental as units of composition in 2-d. But, more importantly, triangles also work when you embed them in higher dimensions, and quadrilaterals don't. You can take any three points and there is a unique flat triangle that they span (it may be degenerate, if the points are collinear). But four points will, in general, span a warped quadrilateral. Mind you, rectangles work great on flat screens, and we use them all the time for selecting things with the mouse. But on a curved or bumpy surface, triangles are the only option.

Surveyors have covered the whole Earth, mountains and all, with triangles. In computer games, we build complex models, including human faces or dolphins, using wireframes. Wireframes are just systems of triangles that share some of the vertices and edges. So triangles can be used to approximate complex 2-d surfaces in 3-d.

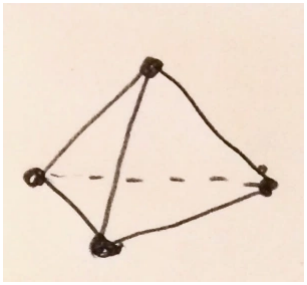


More dimensions

How can we generalize this process? First of all, we could use triangles in spaces that have more than 3 dimensions. This way we could, for instance, build a Klein bottle in 4-d without it intersecting itself.



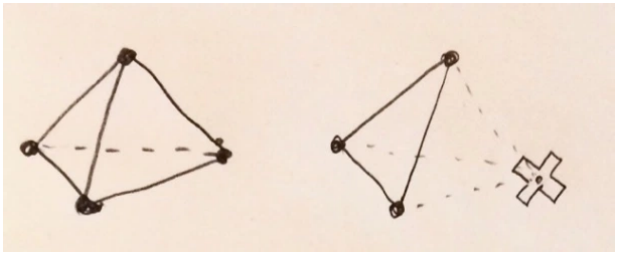
We can also consider replacing triangles with higher-dimensional objects. For instance, we could approximate 3-d volumes by filling them with cubes. This technique is used in computer graphics, where we often organize lots of cubes in data structures called octrees. But just like squares or quadrilaterals don't work very well on non-flat surfaces, cubes cannot be used in curved spaces. The natural generalization of a triangle to something that can fill a volume without any warping is a tetrahedron. Any four points in space span a tetrahedron.



We can go on generalizing this construction to higher and higher dimensions. To form an n-dimensional simplex we can pick $n+1$ points. We can draw a segment between any two points, a triangle between any three points, a tetrahedron between any four points, and so on. It's thus natural to define a 1-dimensional simplex to be a segment, and a 0-dimensional simplex to be a point.

Simplexes (or simplices, as they are sometimes called) have very regular recursive structure. An n-dimensional simplex has $n+1$ faces, which are all $n-1$ dimensional simplexes. A tetrahedron has four triangular faces, a triangle has three sides (one-dimensional simplexes), and a segment has two endpoints. (A point should have one face–and it does, in the "augmented" theory). Every higher-dimensional simplex can be decomposed into lower-dimensional simplexes, and the process can be repeated until we get down to individual vertexes. This constitutes a very interesting composition scheme that will come up over and over again in unexpected places.

Notice that you can always construct a face of a simplex by deleting one point. It's the point opposite to the face in question. This is why there are as many faces as there are points in a simplex.



Look Ma! No coordinates!

So far we've been secretly thinking of points as elements of some n-dimensional linear space, presumably $\mathbb{R}^n$. Time to make another leap of abstraction. Let's abandon coordinate systems. Can we still define simplexes and, if so, how would we use them?

Consider a wireframe built from triangles. It defines a particular shape. We can deform this shape any way we want but, as long as we don't break connections or fuse points, we cannot change its topology. A wireframe corresponding to a torus can never be deformed into a wireframe corresponding to a sphere.

The information about topology is encoded in connections. The connections don't depend on coordinates. Two points are either connected or not. Two triangles either share a side or they don't. Two tetrahedrons either share a triangle or they don't. So if we can define simplexes without resorting to coordinates, we'll have a new language to talk about topology.

But what becomes of a point if we discard its coordinates? It becomes an element of a set. An arrangement of simplexes can be built from a set of points or 0-simplexes, together with a set of 1-simplexes, a set of 2-simplexes, and so on. Imagine that you bought a piece of furniture from Ikea. There is a bag of screws (0-simplexes), a box of sticks (1-simplexes), a crate of triangular planks (2-simplexes), and so on. All parts are freely stretchable (we don't care about sizes).

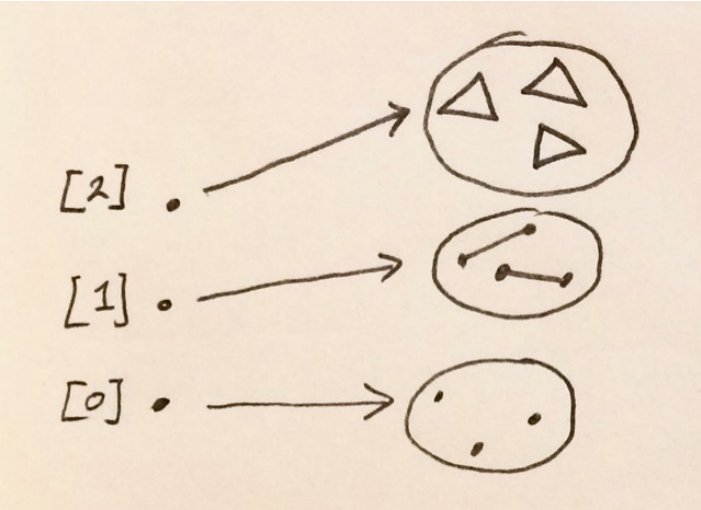
You have no idea what the piece of furniture will look like unless you have an instruction booklet. The booklet tells you how to arrange things: which sticks form the edges of which triangles, etc. In general, you want to know which lower-order simplexes are the "faces" of higher-order simplexes. This can be determined by defining functions between the corresponding sets, which we'll call *face maps*.

For instance, there should be two function from the set of segments to the set of points; one assigning the beginning, and the other the end, to each segment. There should be three functions from the set of triangles to the set of segments, and so on. If the same point is the end of one segment and the beginning of another, the two segments are connected. A segment may be shared between multiple triangles, a triangle may be shared between tetrahedrons, and so on.

You can compose these functions–for instance, to select a vertex of a triangle, or a side of a tetrahedron. Composable functions suggest a category, in this case a subcategory of Set. Selecting a subcategory suggests a *functor* from some other, simpler, category. What would that category be?

The Simplicial category

The objects of this simpler category, let's call it the *simplicial category* Δ, would be mapped by our functor to corresponding sets of simplexes in Set. So, in Δ, we need one object corresponding to the set of points, let's call it $[0]$; another for segments, $[1]$; another for triangles, $[2]$; and so on. In other words, we need one object called $[n]$ per one set of n-dimensional simplexes.



What really determines the structure of this category is its morphisms. In particular, we need morphisms that would be mapped, under our functor, to the functions that define faces of our simplexes–the face maps. This means, in particular, that for every n we need $n+1$ distinct functions from the image of $[n]$ to the image of $[n-1]$. These functions are themselves images of morphisms that go *between* $[n]$ and $[n-1]$ in Δ; we do, however, have a choice of the *direction* of these morphisms. If we choose our functor to be *contravariant*, the face maps from the image of $[n]$ to the image of $[n-1]$ will be images of morphisms going from $[n-1]$ to $[n]$ (the opposite direction). This contravariant functor from Δ to Set (such functors are called pre-sheafe) is called the *simplicial set*.

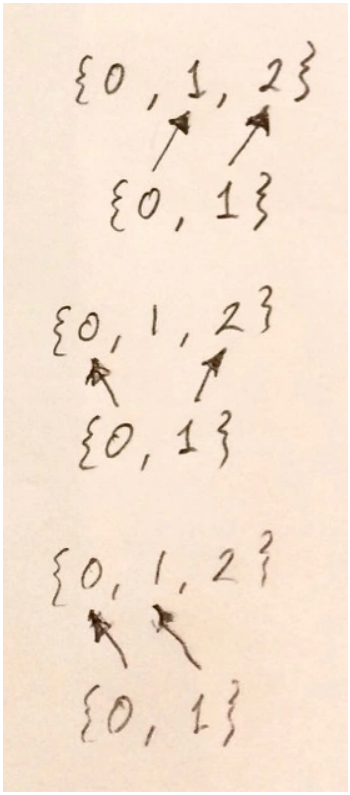
What's attractive about this idea is that there is a category that has exactly the right types of morphisms. It's a category whose objects are ordinals, or ordered sets of numbers, and morphisms are order-preserving functions. Object $[0]$ is the one-element set $\{0\}$, $[1]$ is the set $\{0, 1\}$, $[2]$ is $\{0, 1, 2\}$, and so on. Morphisms are functions that preserve order, that is, if $n < m$ then $f(n) \leq f(m)$. Notice that the inequality is non-strict. This will become important in the definition of degeneracy maps.

The description of simplicial sets using a functor follows a very common pattern in category theory. The simpler category defines the primitives and the grammar for combining them. The target category (often the category of sets) provides models for the theory in question. The same trick is used, for instance, in defining abstract algebras in Lawvere theories. There, too, the syntactic category consists of a tower of objects with a very regular set of morphisms, and the models are contravariant Set-valued functors.

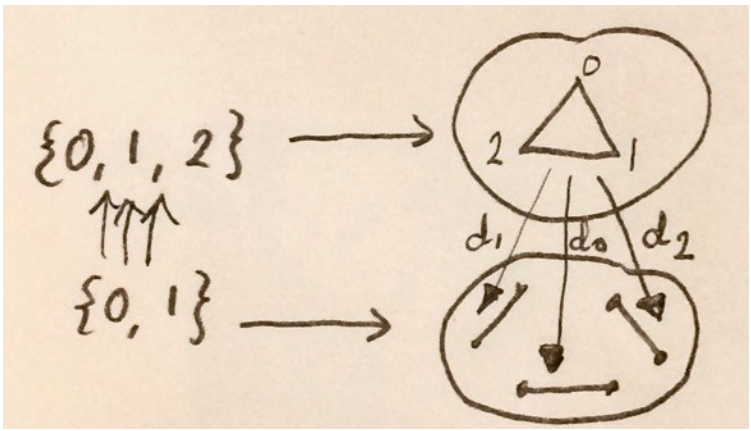
Because simplicial sets are functors, they form a functor category, with natural transformations as morphisms. A natural transformation between two simplicial sets is a family of functions that map vertices to vertices, edges to edges, triangles to triangles, and so on. In other words, it embeds one simplicial set in another.

Face maps

We will obtain face maps as images of *injective* morphisms between objects of Δ. Consider, for instance, an injection from $[1]$ to $[2]$. Such a morphism takes the set $\{0, 1\}$ and maps it to $\{0, 1, 2\}$. In doing so, it must skip one of the numbers in the second set, preserving the order of the other two. There are exactly three such morphisms, skipping either 0, 1, or 2.



And, indeed, they correspond to three face maps. If you think of the three numbers as numbering the vertices of a triangle, the three face maps remove the skipped vertex from the triangle leaving the opposing side free. The functor is contravariant, so it reverses the direction of morphisms.



The same procedure works for higher order simplexes. An injection from $[n-1]$ to $[n]$ maps $\{0, 1, ..., n-1\}$ to $\{0, 1, ..., n\}$ by skipping some k between 0 and n.

The corresponding face map is called $d_{n,k}$, or simply d_k, if n is obvious from the context.

Such face maps automatically satisfy the obvious identities for any $i < j$:

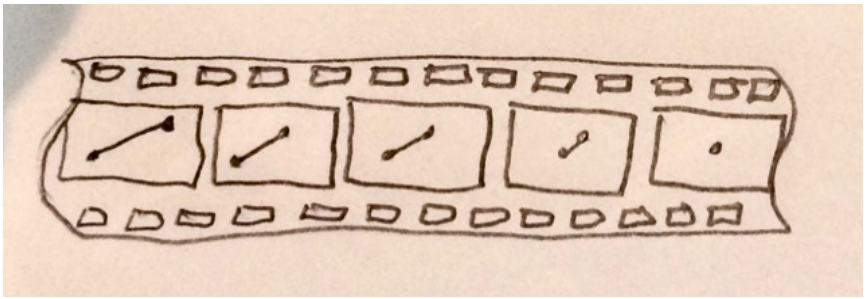
$$d_i d_j = d_{j-1} d_i$$

The change from j to $j-1$ on the right compensates for the fact that, after removing the i th number, the remaining indexes are shifted down.

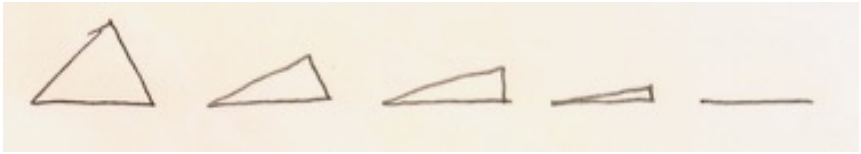
These injections generate, through composition, all the morphisms that strictly preserve the ordering (we also need identity maps to form a category). But, as I mentioned before, we are also interested in those maps that are non-strict in the preservation of ordering (that is, they can map two consecutive numbers into one). These generate the so called *degeneracy maps*. Before we get to definitions, let me provide some motivation.

Homotopy

One of the important application of simplexes is in homotopy. You don't need to study algebraic topology to get a feel of what homotopy is. Simply said, homotopy deals with shrinking and holes. For instance, you can always shrink a segment to a point. The intuition is pretty obvious. You have a segment at time zero, and a point at time one, and you can create a continuous "movie" in between. Notice that a segment is a 1-simplex, whereas a point is a 0-simplex. Shrinking therefore provides a bridge between different-dimensional simplexes.



Similarly, you can shrink a triangle to a segment–in particular the segment that is one of its sides.



You can also shrink a triangle to a point by pasting together two shrinking movies–first shrinking the triangle to a segment, and then the segment to a point. So shrinking is composable.

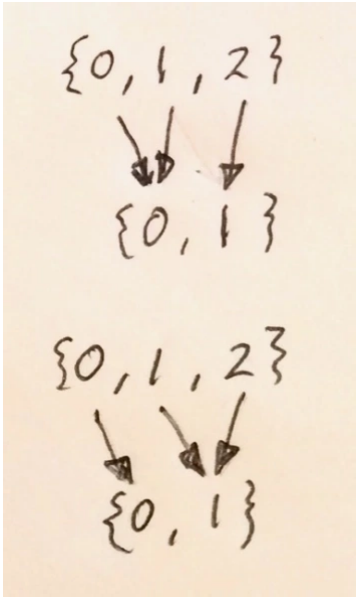
But not all higher-dimensional shapes can be shrunk to all lower-dimensional shapes. For instance, an annulus (a.k.a., a ring) cannot be shrunk to a segment–this would require tearing it. It can, however, be shrunk to a circular loop (or two segments connected end to end to form a loop). That's because both, the annulus and the circle, have a hole. So continuous shrinking can be used to classify shapes according to how many holes they have.

We have a problem, though: You can't describe continuous transformations without using coordinates. But we can do the next best thing: We can define degenerate simplexes to bridge the gap between dimensions. For instance, we can build a segment, which uses the same vertex twice. Or a collapsed triangle, which uses the same side twice (its third side is a degenerate segment).

Degeneracy maps

We model operations on simplexes, such as face maps, through morphisms from the category opposite to Δ. The creation of degenerate simplexes will therefore corresponds to mappings from $[n+1]$ to $[n]$. They obviously cannot be injective, but we may chose them to be surjective. For instance, the creation of a degenerate segment from a point corresponds to the (opposite) mapping of $\{0,1\}$ to $\{0\}$, which collapses the two numbers to one.

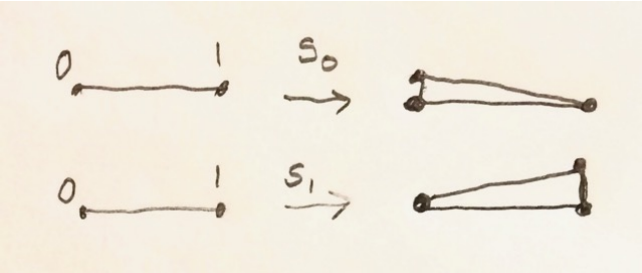
We can construct a degenerate triangle from a segment in two ways. These correspond to the two surjections from $\{0,1,2\}$ to $\{0,1\}$.



The first one called $\sigma_{1,0}$ maps both 0 and 1 to 0 and 2 to 1. Notice that, as required, it preserves the order, albeit weakly. The second, $\sigma_{1,1}$ maps 0 to 0 but collapses 1 and 2 to 1.

In general, $\sigma_{n,k}$ maps $\{0,1,...k,k+1...n+1\}$ to $\{0,1,...k...n\}$ by collapsing k and $k+1$ to k.

Our contravariant functor maps these order-preserving surjections to functions on sets. The resulting functions are called *degeneracy maps*: each $\sigma_{n,k}$ mapped to the corresponding $s_{n,k}$. As with face maps, we usually omit the first index, as it's either arbitrary or easily deducible from the context.



Two degeneracy maps. In the triangles, two of the sides are actually the same segment. The third side is a degenerate segment whose ends are the same point.

There is an obvious identity for the composition of degeneracy maps:

$$s_i s_j = s_{j+1} s_i$$

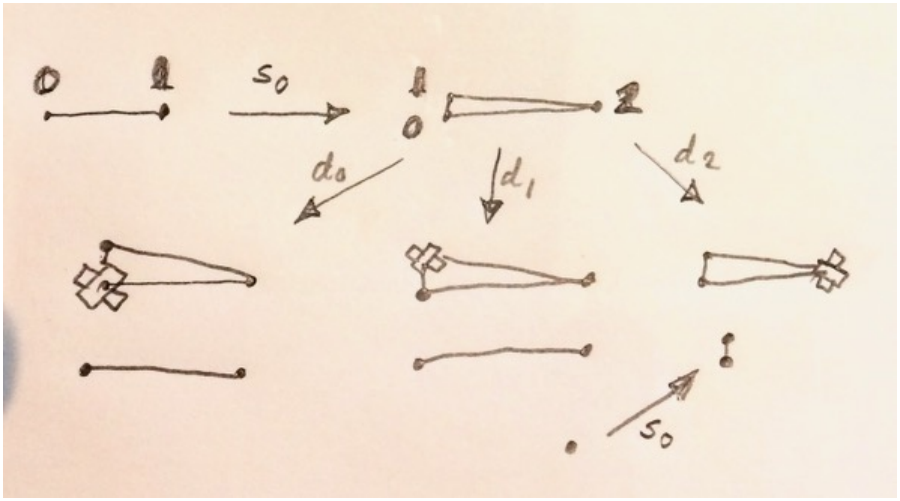
for $i \leq j$.

The interesting identities relate degeneracy maps to face maps. For instance, when $i = j$ or $i = j + 1$, we have:

$$d_i s_j = id$$

(that's the identity morphism). Geometrically speaking, imagine creating a degenerate triangle from a segment, for instance by using s_0. The first side of this triangle, which is obtained by applying d_0, is the original segment. The second side, obtained by d_1, is the same segment again.

The third side is degenerate: it can be obtained by applying s_0 to the vertex obtained by d_1.



In general, for $i > j + 1$:

$$d_i s_j = s_j d_{i-1}$$

Similarly:

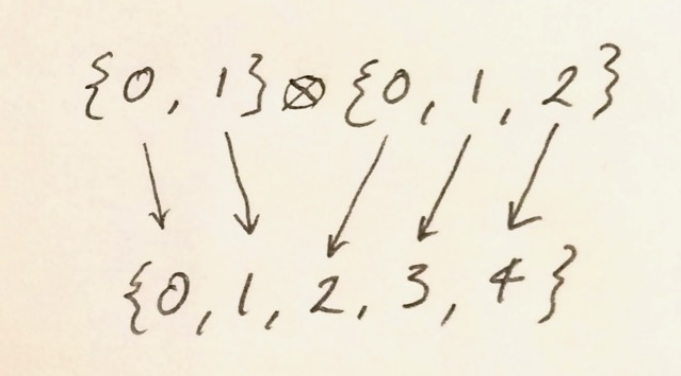
$$d_i s_j = s_{j-1} d_i$$

for $i < j$.

All the face- and degeneracy-map identities are relevant because, given a family of sets and functions that satisfy them, we can reproduce the simplicial set (contravariant functor from Δ to Set) that generates them. This shows the equivalence of the geometric picture that deals with triangles, segments, faces, etc., with the combinatorial picture that deals with rearrangements of ordered sequences of numbers.

Monoidal structure

A triangle can be constructed by adjoining a point to a segment. Add one more point and you get a tetrahedron. This process of adding points can be extended to adding together arbitrary simplexes. Indeed, there is a binary operator in Δ that combines two ordered sequences by stacking one after another.



This operation can be lifted to morphisms, making it a bifunctor. It is associative, so one might ask the question whether it can be used as a tensor product to make Δ a monoidal category. The only thing missing is the unit object.

The lowest dimensional simplex in Δ is $[0]$, which represents a point, so it cannot be a unit with respect to our tensor product. Instead we are obliged to add a new object, which is called $[-1]$, and is represented by an empty set. (Incidentally, this is the object that may serve as "the face" of a point.)

With the new object $[-1]$, we get the category Δ_a, which is called the augmented simplicial category. Since the unit and associativity laws are satisfied "on the nose" (as opposed to "up to isomorphism"), Δ_a is a *strict* monoidal category.

Note: Some authors prefer to name the objects of Δ_a starting from zero, rather than minus one. They rename $[-1]$ to $\mathbf{0}$, $[0]$ to $\mathbf{1}$, etc. This convention makes even more sense if you consider that $\mathbf{0}$ is the initial object and $\mathbf{1}$ the terminal object in Δ_a.

Monoidal categories are a fertile breeding ground for monoids. Indeed, the object $[0]$ in Δ_a is a monoid. It is equipped with two morphisms that act like unit and multiplication. It has an incoming morphism from the monoidal unit $[-1]$–the morphism that's the precursor of the face map that assigns the empty set to every point. This morphism can be used as the unit η of our monoid. It also has an incoming morphism from $[1]$ (which happens to be the tensorial square of $[0]$). It's the precursor of the degeneracy map that creates a segment from a single point. This morphism is the multiplication μ of our monoid. Unit and associativity laws follow from the standard identities between morphisms in Δ_a.

It turns out that this monoid $([0], \eta, \mu)$ in Δ_a is the mother of all monoids in strict monoidal categories. It can be shown that, for any monoid m in any strict monoidal category C, there is a *unique* strict monoidal functor F from Δ_a to C that maps the monoid $[0]$ to the monoid m. The category Δ_a has exactly the right structure, and nothing more, to serve as the pattern for any monoid we can come up within a (strict) monoidal category. In particular, since a monad is just a monoid in the (strictly monoidal) category of endofunctors, the augmented simplicial category is behind every monad as well.

One more thing

Incidentally, since Δ_a is a monoidal category, (contravariant) functors from it to Set are automatically equipped with monoidal structure via Day convolution. The result of Day convolution is a *join* of simplicial sets. It's a generalized cone: two simplicial sets together with all possible connections between them. In particular, if one of the sets is just a single point, the result of the join is an actual cone (or a pyramid).

Different shapes

If we are willing to let go of geometric interpretations, we can replace the target category of sets with an arbitrary category. Instead of having a set of simplexes, we'll end up with an object of simplexes. Simplicial sets become *simplicial objects*.

Alternatively, we can generalize the source category. As I mentioned before, simplexes are a good choice of primitives because of their geometrical properties–they don't warp. But if we don't care about embedding these simplexes in $\mathbb{R}^n$, we can replace them with cubes of varying dimensions (a one dimensional cube is a segment, a two dimensional cube is a square, and so on). Functors from the category of n-cubes to Set are called *cubical sets*. An even further generalization replaces simplexes with shapeless globes producing *globular sets*.

All these generalizations become important tools in studying higher category theory. In an n-category, we naturally encounter various shapes, as reflected in the naming convention: objects are called 0-cells; morphisms, 1-cells; morphisms between morphisms, 2-cells, and so on. These "cells" are often visualized as n-dimensional shapes. If a 1-cell is an arrow, a 2-cell is a (directed) surface spanning two arrows; a 3-cell, a volume between two surfaces; e.t.c. In this way, the shapeless hom-set that connects two objects in a regular category turns into a topologically rich blob in an n-category.

This is even more pronounced in infinity groupoids, which became popularized by homotopy type theory, where we have an infinite tower of bidirectional n-morphisms. The presence or the absence of higher order morphisms between any two morphisms can be visualized as the existence of holes that prevent the morphing of one cell into another. This kind of morphing can be described by homotopies which, in turn, can be described using simplicial, cubical, globular, or even more exotic sets.

Conclusion

I realize that this post might seem a little rambling. I have two excuses: One is that, when I started looking at simplexes, I had no idea where I would end up. One thing led to another and I was totally fascinated by the journey. The other is the realization how everything is related to everything else in mathematics. You start with simple triangles, you compose and decompose them, you see some structure emerging. Suddenly, the same compositional structure pops up in totally unrelated areas. You see it in algebraic topology, in a monoid in a monoidal category, or in a generalization of a hom-set in an n-category. Why is it so? It seems like there aren't that many ways of composing things together, and we are forced to keep reusing them over and over again. We can glue them, nail them, or solder them. The way simplicial category is put together provides a template for one of the universal patterns of composition.

Bibliography

1. John Baez, A Quick Tour of Basic Concepts in Simplicial Homotopy Theory
2. Greg Friedman, An elementary illustrated introduction to simplicial sets.
3. N J Wildberger, Algebraic Topology. An excellent series of videos.

Acknowledgments

I'm grateful to Edward Kmett and Derek Elkins for reviewing the draft and for providing helpful suggestions.

October 12, 2018

Trading FunLists at a Bazaar with Yoneda

Posted by Bartosz Milewski under Category Theory, Functional Programming, Haskell, Programming [5] Comments

i
5 Votes

There is a lot of folklore about various data types that pop up in discussions about lenses. For instance, it's known that `FunList` and `Bazaar` are equivalent, although I haven't seen a proof of that. Since both data structures appear in the context of `Traversable`, which is of great interest to me, I decided to do some research. In particular, I was interested in translating these data structures into constructs in category theory. This is a continuation of my previous blog posts on free monoids and free applicatives. Here's what I have found out:

- `FunList` is a free applicative generated by the `Store` functor. This can be shown by expressing the free applicative construction using Day convolution.
- Using Yoneda lemma in the category of applicative functors I can show that `Bazaar` is equivalent to `FunList`

Let's start with some definitions. `FunList` was first introduced by Twan van Laarhoven in his blog. Here's a (slightly generalized) Haskell definition:

```
data FunList a b t = Done t
                   | More a (FunList a b (b -> t))
```

It's a non-regular inductive data structure, in the sense that its data constructor is recursively called with a different type, here the function type `b->t`. `FunList` is a functor in `t`, which can be written categorically as:

$$L_{ab}t = t + a \times L_{ab}(b \to t)$$

where $b \to t$ is a shorthand for the hom-set $Set(b, t)$.

Strictly speaking, a recursive data structure is defined as an initial algebra for a higher-order functor. I will show that the higher order functor in question can be written as:

$$A_{ab}g = I + \sigma_{ab} \star g$$

where σ_{ab} is the (indexed) store comonad, which can be written as:

$$\sigma_{ab}s = \Delta_a s \times C(b, s)$$

Here, Δ_a is the constant functor, and $C(b, -)$ is the hom-functor. In Haskell, this is equivalent to:

```
newtype Store a b s = Store (a, b -> s)
```

The standard (non-indexed) `Store` comonad is obtained by identifying `a` with `b` and it describes the objects of the slice category C/s (morphisms are functions $f : a \to a'$ that make the obvious triangles commute).

If you've read my previous blog posts, you may recognize in A_{ab} the functor that generates a free applicative functor (or, equivalently, a free monoidal functor). Its fixed point can be written as:

$$L_{ab} = I + \sigma_{ab} \star L_{ab}$$

The star stands for Day convolution–in Haskell expressed as an existential data type:

```
data Day f g s where
  Day :: f a -> g b -> ((a, b) -> s) -> Day f g s
```

Intuitively, L_{ab} is a "list of" `Store` functors concatenated using Day convolution. An empty list is the identity functor, a one-element list is the `Store` functor, a two-element list is the Day convolution of two `Store` functors, and so on…

In Haskell, we would express it as:

```
data FunList a b t = Done t
                   | More ((Day (Store a b) (FunList a b)) t)
```

To show the equivalence of the two definitions of `FunList`, let's expand the definition of Day convolution inside A_{ab}:

$$(A_{ab}g)t = t + \int^{cd} (\Delta_b c \times C(a, c)) \times gd \times C(c \times d, t)$$

The coend $\int^{cd}$ corresponds, in Haskell, to the existential data type we used in the definition of `Day`.

Since we have the hom-functor $C(a, c)$ under the coend, the first step is to use the co-Yoneda lemma to "perform the integration" over c, which replaces c with a everywhere. We get:

$$t + \int^{d} \Delta_b a \times gd \times C(a \times d, t)$$

We can then evaluate the constant functor and use the currying adjunction:

$$C(a \times d, t) \cong C(d, a \to t)$$

to get:

$$t + \int^{d} b \times gd \times C(d, a \to t)$$

Applying the co-Yoneda lemma again, we replace d with $a \to t$:

$$t + b \times g(a \to t)$$

This is exactly the functor that generates `FunList`. So `FunList` is indeed the free applicative generated by `Store`.

All transformations in this derivation were natural isomorphisms.

Now let's switch our attention to `Bazaar`, which can be defined as:

```
type Bazaar a b t = forall f. Applicative f => (a -> f b) -> f t
```

(The actual definition of `Bazaar` in the lens library is even more general–it's parameterized by a profunctor in place of the arrow in `a -> f b`.)

The universal quantification in the definition of `Bazaar` immediately suggests the application of my favorite double Yoneda trick in the functor category: The set of natural transformations (morphisms in the functor category) between two functors (objects in the functor category) is isomorphic, through Yoneda embedding, to the following end in the functor category:

$$Nat(h, g) \cong \int_{f\colon [C,Set]} Set(Nat(g, f), Nat(h, f))$$

The end is equivalent (modulo parametricity) to Haskell `forall`. Here, the sets of natural transformations between pairs of functors are just hom-functors in the functor category and the end over f is a set of higher-order natural transformations between them.

In the double Yoneda trick we carefully select the two functors g and h to be either representable, or somehow related to representables.

The universal quantification in `Bazaar` is limited to applicative functors, so we'll pick our two functors to be free applicatives. We've seen previously that the higher-order functor that generates free applicatives has the form:

$$Fg = Id + g \star Fg$$

Here's the version of the Yoneda embedding in which f varies over all applicative functors in the category App, and g and h are arbitrary functors in $[C, Set]$:

$$App(Fh, Fg) \cong \int_{f\colon App} Set(App(Fg, f), App(Fh, f))$$

The free functor F is the left adjoint to the forgetful functor U:

$$App(Fg, f) \cong [C, Set](g, Uf)$$

Using this adjunction, we arrive at:

$$[C, Set](h, U(Fg)) \cong \int_{f\colon App} Set([C, Set](g, Uf), [C, Set](h, Uf))$$

We're almost there–we just need to carefuly pick the functors g and h. In order to arrive at the definition of `Bazaar` we want:

$$g = \sigma_{ab} = \Delta_a \times C(b, -)$$

$$h = C(t, -)$$

The right hand side becomes:

$$\int_{f\colon App} Set\big(\int_c Set(\Delta_a c \times C(b, c), (Uf)c)), \int_c Set(C(t, c), (Uf)c)\big)$$

where I represented natural transformations as ends. The first term can be curried:

$$Set\big(\Delta_a c \times C(b, c), (Uf)c)\big) \cong Set\big(C(b, c), \Delta_a c \to (Uf)c\big)$$

and the end over c can be evaluated using the Yoneda lemma. So can the second term. Altogether, the right hand side becomes:

$$\int_{f\colon App} Set\big(a \to (Uf)b)), (Uf)t)\big)$$

In Haskell notation, this is just the definition of `Bazaar`:

```
forall f. Applicative f => (a -> f b) -> f t
```

The left hand side can be written as:

$$\int_c Set(hc, (U(Fg))c)$$

Since we have chosen h to be the hom-functor $C(t, -)$, we can use the Yoneda lemma to "perform the integration" and arrive at:

$$(U(Fg))t$$

With our choice of $g = \sigma_{ab}$, this is exactly the free applicative generated by `Store`–in other words, `FunList`.

This proves the equivalence of `Bazaar` and `FunList`. Notice that this proof is only valid for Set-valued functors, although a generalization to the enriched setting is relatively straightforward.

There is another family of functors, `Traversable`, that uses universal quantification over applicatives:

```
class (Functor t, Foldable t) => Traversable t where
  traverse :: forall f. Applicative f => (a -> f b) -> t a -> f (t b)
```

The same double Yoneda trick can be applied to it to show that it's related to `Bazaar`. There is, however, a much simpler derivation, suggested to me by Derek Elkins, by changing the order of arguments:

```
traverse :: t a -> (forall f. Applicative f => (a -> f b) -> f (t b))
```

which is equivalent to:

```
traverse :: t a -> Bazaar a b (t b)
```

In view of the equivalence between `Bazaar` and `FunList`, we can also write it as:

```
traverse :: t a -> FunList a b (t b)
```

Note that this is somewhat similar to the definition of `toList`:

```
toList :: Foldable t => t a -> [a]
```

In a sense, `FunList` is able to freely accumulate the effects from `traversable`, so that they can be interpreted later.

Acknowledgments

I'm grateful to Edward Kmett and Derek Elkins for many discussions and valuable insights.

August 20, 2018

Recursion Schemes for Higher Algebras

Posted by Bartosz Milewski under Category Theory, Functional Programming, Haskell, Monads, Programming
[2] Comments

i
14 Votes

Abstract

The use of free monads, free applicatives, and cofree comonads lets us separate the construction of (often effectful or context-dependent) computations from their interpretation. In this paper I show how the ad hoc process of writing interpreters for these free constructions can be systematized using the language of higher order algebras (coalgebras) and catamorphisms (anamorphisms).

Introduction

Recursive schemes [meijer] are an example of successful application of concepts from category theory to programming. The idea is that recursive data structures can be defined as initial algebras of functors. This allows a separation of concerns: the functor describes the local shape of the data structure, and the

fixed point combinator builds the recursion. Operations over data structures can be likewise separated into shallow, non-recursive computations described by algebras, and generic recursive procedures described by catamorphisms. In this way, data structures often replace control structures in driving computations.

Since functors also form a category, it's possible to define functors acting on functors. Such higher order functors show up in a number of free constructions, notably free monads, free applicatives, and cofree comonads. These free constructions have good composability properties and they provide means of separating the creation of effectful computations from their interpretation.

This paper's contribution is to systematize the construction of such interpreters. The idea is that free constructions arise as fixed points of higher order functors, and therefore can be approached with the same algebraic machinery as recursive data structures, only at a higher level. In particular, interpreters can be constructed as catamorphisms or anamorphisms of higher order algebras/coalgebras.

Initial Algebras and Catamorphisms

The canonical example of a data structure that can be described as an initial algebra of a functor is a list. In Haskell, a list can be defined recursively:

```
data List a = Nil | Cons a (List a)
```

There is an underlying non-recursive functor:

```
data ListF a x = NilF | ConsF a x
instance Functor (ListF a) where
  fmap f NilF = NilF
  fmap f (ConsF a x) = ConsF a (f x)
```

Once we have a functor, we can define its algebras. An *algebra* consist of a carrier `c` and a structure map (evaluator). An algebra can be defined for an arbitrary functor `f`:

```
type Algebra f c = f c -> c
```

Here's an example of a simple list algebra, with `Int` as its carrier:

```
sum :: Algebra (ListF Int) Int
sum NilF = 0
sum (ConsF a c) = a + c
```

Algebras for a given functor form a category. The initial object in this category (if it exists) is called the initial algebra. In Haskell, we call the carrier of the initial algebra `Fix f`. Its structure map is a function:

```
f (Fix f) -> Fix f
```

By Lambek's lemma, the structure map of the initial algebra is an isomorphism. In Haskell, this isomorphism is given by a pair of functions: the constructor `In` and the destructor `out` of the fixed point combinator:

```
newtype Fix f = In { out :: f (Fix f) }
```

When applied to the list functor, the fixed point gives rise to an alternative definition of a list:

```
type List a = Fix (ListF a)
```

The initiality of the algebra means that there is a unique algebra morphism from it to any other algebra. This morphism is called a *catamorphism* and, in Haskell, can be expressed as:

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

A list catamorphism is known as a fold. Since the list functor is a sum type, its algebra consists of a value —the result of applying the algebra to `NilF`—and a function of two variables that corresponds to the `ConsF` constructor. You may recognize those two as the arguments to `foldr`:

```
foldr :: (a -> c -> c) -> c -> [a] -> c
```

The list functor is interesting because its fixed point is a free monoid. In category theory, monoids are special objects in monoidal categories—that is categories that define a product of two objects. In Haskell, a pair type plays the role of such a product, with the unit type as its unit (up to isomorphism).

As you can see, the list functor is the sum of a unit and a product. This formula can be generalized to an arbitrary monoidal category with a tensor product $\otimes$ and a unit 1:

$$L\,a\,x = 1 + a \otimes x$$

Its initial algebra is a free monoid .

Higher Algebras

In category theory, once you performed a construction in one category, it's easy to perform it in another category that shares similar properties. In Haskell, this might require reimplementing the construction.

We are interested in the category of endofunctors, where objects are endofunctors and morphisms are natural transformations. Natural transformations are represented in Haskell as polymorphic functions:

```
type f :~> g = forall a. f a -> g a
infixr 0 :~>
```

In the category of endofunctors we can define (higher order) functors, which map functors to functors and natural transformations to natural transformations:

```
class HFunctor hf where
  hfmap :: (g :~> h) -> (hf g :~> hf h)
  ffmap :: Functor g => (a -> b) -> hf g a -> hf g b
```

The first function lifts a natural transformation; and the second function, `ffmap`, witnesses the fact that the result of a higher order functor is again a functor.

An algebra for a higher order functor `hf` consists of a functor `f` (the carrier object in the functor category) and a natural transformation (the structure map):

```
type HAlgebra hf f = hf f :~> f
```

As with regular functors, we can define an initial algebra using the fixed point combinator for higher order functors:

```
newtype FixH hf a = InH { outH :: hf (FixH hf) a }
```

Similarly, we can define a higher order catamorphism:

```
hcata :: HFunctor h => HAlgebra h f -> FixH h :~> f
hcata halg = halg . hfmap (hcata halg) . outH
```

The question is, are there any interesting examples of higher order functors and algebras that could be used to solve real-life programming problems?

Free Monad

We've seen the usefulness of lists, or free monoids, for structuring computations. Let's see if we can generalize this concept to higher order functors.

The definition of a list relies on the cartesian structure of the underlying category. It turns out that there are multiple cartesian structures of interest that can be defined in the category of functors. The simplest one defines a product of two endofunctors as their composition. Any two endofunctors can be composed. The unit of functor composition is the identity functor.

If you picture endofunctors as containers, you can easily imagine a tree of lists, or a list of `Maybe`s.

A monoid based on this particular monoidal structure in the endofunctor category is a monad. It's an endofunctor `m` equipped with two natural transformations representing unit and multiplication:

```
class Monad m where
  eta :: Identity    :~> m
  mu  :: Compose m m :~> m
```

In Haskell, the components of these natural transformations are known as `return` and `join`.

A straightforward generalization of the list functor to the functor category can be written as:

$$L\ f\ g = 1 + f \circ g$$

or, in Haskell,

```
type FunctorList f g = Identity :+: Compose f g
```

where we used the operator `:+:` to define the coproduct of two functors:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
infixr 7 :+:
```

Using more conventional notation, `FunctorList` can be written as:

```
data MonadF f g a =
    DoneM a
  | MoreM (f (g a))
```

We'll use it to generate a free monoid in the category of endofunctors. First of all, let's show that it's indeed a higher order functor in the second argument `g`:

```
instance Functor f => HFunctor (MonadF f) where
  hfmap _   (DoneM a)  = DoneM a
  hfmap nat (MoreM fg) = MoreM $ fmap nat fg
  ffmap h (DoneM a)    = DoneM (h a)
  ffmap h (MoreM fg)   = MoreM $ fmap (fmap h) fg
```

In category theory, because of size issues, this functor doesn't always have a fixed point. For most common choices of `f` (e.g., for algebraic data types), the initial higher order algebra for this functor exists, and it generates a free monad. In Haskell, this free monad can be defined as:

```
type FreeMonad f = FixH (MonadF f)
```

We can show that `FreeMonad` is indeed a monad by implementing `return` and `bind`:

```
instance Functor f => Monad (FreeMonad f) where
  return = InH . DoneM
  (InH (DoneM a))    >>= k = k a
  (InH (MoreM ffra)) >>= k =
        InH (MoreM (fmap (>>= k) ffra))
```

Free monads have many applications in programming. They can be used to write generic monadic code, which can then be interpreted in different monads. A very useful property of free monads is that they can be composed using coproducts. This follows from the theorem in category theory, which states that left adjoints preserve coproducts (or, more generally, colimits). Free constructions are, by definition, left adjoints to forgetful functors. This property of free monads was explored by Swierstra [swierstra] in his solution to the expression problem. I will use an example based on his paper to show how to construct monadic interpreters using higher order catamorphisms.

Free Monad Example

A stack-based calculator can be implemented directly using the state monad. Since this is a very simple example, it will be instructive to re-implement it using the free monad approach.

We start by defining a functor, in which the free parameter `k` represents the continuation:

```
data StackF k  = Push Int k
               | Top (Int -> k)
               | Pop k
               | Add k
               deriving Functor
```

We use this functor to build a free monad:

```
type FreeStack = FreeMonad StackF
```

You may think of the free monad as a tree with nodes that are defined by the functor `StackF`. The unary constructors, like `Add` or `Pop`, create linear list-like branches; but the `Top` constructor branches out with one child per integer.

The level of indirection we get by separating recursion from the functor makes constructing free monad trees syntactically challenging, so it makes sense to define a helper function:

```
liftF :: (Functor f) => f r -> FreeMonad f r
liftF fr = InH $ MoreM $ fmap (InH . DoneM) fr
```

With this function, we can define smart constructors that build leaves of the free monad tree:

```
push :: Int -> FreeStack ()
push n = liftF (Push n ())

pop :: FreeStack ()
pop = liftF (Pop ())

top :: FreeStack Int
top = liftF (Top id)

add :: FreeStack ()
add = liftF (Add ())
```

All these preparations finally pay off when we are able to create small programs using `do` notation:

```
calc :: FreeStack Int
calc = do
  push 3
  push 4
  add
  x <- top
  pop
  return x
```

Of course, this program does nothing but build a tree. We need a separate interpreter to do the calculation. We'll interpret our program in the state monad, with state implemented as a stack (list) of integers:

```
type MemState = State [Int]
```

The trick is to define a higher order algebra for the functor that generates the free monad and then use a catamorphism to apply it to the program. Notice that implementing the algebra is a relatively simple procedure because we don't have to deal with recursion. All we need is to case-analyze the shallow constructors for the free monad functor `MonadF`, and then case-analyze the shallow constructors for the functor `StackF`.

```
runAlg :: HAlgebra (MonadF StackF) MemState
runAlg (DoneM a)  = return a
runAlg (MoreM ex) =
  case ex of
    Top  ik  -> get >>= ik  . head
    Pop  k   -> get >>= put . tail   >> k
    Push n k -> get >>= put . (n : ) >> k
    Add  k   -> do (a: b: s) <- get
                   put (a + b : s)
                   k
```

The catamorphism converts the program `calc` into a state monad action, which can be run over an empty initial stack:

```
runState (hcata runAlg calc) []
```

The real bonus is the freedom to define other interpreters by simply switching the algebras. Here's an algebra whose carrier is the `Const` functor:

```
showAlg :: HAlgebra (MonadF StackF) (Const String)

showAlg (DoneM a) = Const "Done!"
showAlg (MoreM ex) = Const $
  case ex of
    Push n k ->
      "Push " ++ show n ++ ", " ++ getConst k
    Top ik ->
      "Top, " ++ getConst (ik 42)
    Pop k ->
      "Pop, " ++ getConst k
    Add k ->
      "Add, " ++ getConst k
```

Runing the catamorphism over this algebra will produce a listing of our program:

```
getConst $ hcata showAlg calc

> "Push 3, Push 4, Add, Top, Pop, Done!"
```

Free Applicative

There is another monoidal structure that exists in the category of functors. In general, this structure will work for functors from an arbitrary monoidal category C to Set. Here, we'll restrict ourselves to endofunctors on Set. The product of two functors is given by Day convolution, which can be implemented in Haskell using an existential type:

```
data Day f g c where
  Day :: f a -> g b -> ((a, b) -> c) -> Day f g c
```

The intuition is that a Day convolution contains a container of some `a`s, and another container of some `b`s, together with a function that can convert any pair `(a, b)` to `c`.

Day convolution is a higher order functor:

```
instance HFunctor (Day f) where
  hfmap nat (Day fx gy xyt) = Day fx (nat gy) xyt
  ffmap h   (Day fx gy xyt) = Day fx gy (h . xyt)
```

In fact, because Day convolution is symmetric up to isomorphism, it is automatically functorial in both arguments.

To complete the monoidal structure, we also need a functor that could serve as a unit with respect to Day convolution. In general, this would be the the hom-functor from the monoidal unit:

$C(1, -)$

In our case, since 1 is the singleton set, this functor reduces to the identity functor.

We can now define monoids in the category of functors with the monoidal structure given by Day convolution. These monoids are equivalent to lax monoidal functors which, in Haskell, form the class:

```
class Functor f => Monoidal f where
  unit  :: f ()
  (>*<) :: f x -> f y -> f (x, y)
```

Lax monoidal functors are equivalent to applicative functors [mcbride], as seen in this implementation of `pure` and `<*>`:

```
  pure  :: a -> f a
  pure a = fmap (const a) unit
  (<*>) :: f (a -> b) -> f a -> f b
  fs <*> as = fmap (uncurry ($)) (fs >*< as)
```

We can now use the same general formula, but with Day convolution as the product:

$$L\ f\ g = 1 + f \star g$$

to generate a free monoidal (applicative) functor:

```
data FreeF f g t =
      DoneF t
    | MoreF (Day f g t)
```

This is indeed a higher order functor:

```
instance HFunctor (FreeF f) where
    hfmap _ (DoneF x)     = DoneF x
    hfmap nat (MoreF day) = MoreF (hfmap nat day)
    ffmap f (DoneF x)     = DoneF (f x)
    ffmap f (MoreF day)   = MoreF (ffmap f day)
```

and it generates a free applicative functor as its initial algebra:

```
type FreeA f = FixH (FreeF f)
```

Free Applicative Example

The following example is taken from the paper by Capriotti and Kaposi [capriotti]. It's an option parser for a command line tool, whose result is a user record of the following form:

```
data User = User
  { username :: String
  , fullname :: String
  , uid      :: Int
  } deriving Show
```

A parser for an individual option is described by a functor that contains the name of the option, an optional default value for it, and a reader from string:

```
data Option a = Option
  { optName    :: String
  , optDefault :: Maybe a
  , optReader  :: String -> Maybe a
  } deriving Functor
```

Since we don't want to commit to a particular parser, we'll create a parsing action using a free applicative functor:

```
userP :: FreeA Option User
userP  = pure User
  <*> one (Option "username" (Just "John")  Just)
  <*> one (Option "fullname" (Just "Doe")   Just)
  <*> one (Option "uid"      (Just 0)       readInt)
```

where `readInt` is a reader of integers:

```
readInt :: String -> Maybe Int
readInt s = readMaybe s
```

and we used the following smart constructors:

```
one :: f a -> FreeA f a
one fa = InH $ MoreF $ Day fa (done ()) fst

done :: a -> FreeA f a
done a = InH $ DoneF a
```

We are now free to define different algebras to evaluate the free applicative expressions. Here's one that collects all the defaults:

```
alg :: HAlgebra (FreeF Option) Maybe
alg (DoneF a) = Just a
alg (MoreF (Day oa mb f)) =
  fmap f (optDefault oa >*< mb)
```

I used the monoidal instance for `Maybe`:

```
instance Monoidal Maybe where
  unit = Just ()
  Just x >*< Just y = Just (x, y)
  _ >*< _ = Nothing
```

This algebra can be run over our little program using a catamorphism:

```
parserDef :: FreeA Option a -> Maybe a
parserDef = hcata alg
```

And here's an algebra that collects the names of all the options:

```
alg2 :: HAlgebra (FreeF Option) (Const String)
alg2 (DoneF a) = Const "."
alg2 (MoreF (Day oa bs f)) =
  fmap f (Const (optName oa) >*< bs)
```

Again, this uses a monoidal instance for `Const`:

```
instance Monoid m => Monoidal (Const m) where
  unit = Const mempty
  Const a >*< Const b = Const (a  b)
```

We can also define the `Monoidal` instance for `IO`:

```
instance Monoidal IO where
  unit = return ()
  ax >*< ay = do a <- ax
                 b <- ay
                 return (a, b)
```

This allows us to interpret the parser in the `IO` monad:

```
alg3 :: HAlgebra (FreeF Option) IO
alg3 (DoneF a) = return a
alg3 (MoreF (Day oa bs f)) = do
    putStrLn $ optName oa
    s <- getLine
    let ma = optReader oa s
        a = fromMaybe (fromJust (optDefault oa)) ma
    fmap f $ return a >*< bs
```

Cofree Comonad

Every construction in category theory has its dual—the result of reversing all the arrows. The dual of a product is a coproduct, the dual of an algebra is a coalgebra, and the dual of a monad is a comonad.

Let's start by defining a higher order coalgebra consisting of a carrier `f`, which is a functor, and a natural transformation:

```
type HCoalgebra hf f = f :~> hf f
```

An initial algebra is dualized to a terminal coalgebra. In Haskell, both are the results of applying the same fixed point combinator, reflecting the fact that the Lambek's lemma is self-dual. The dual to a catamorphism is an anamorphism. Here is its higher order version:

```
hana :: HFunctor hf
     => HCoalgebra hf f -> (f :~> FixH hf)
hana hcoa = InH . hfmap (hana hcoa) . hcoa
```

The formula we used to generate free monoids:

$$1 + a \otimes x$$

dualizes to:

$$1 \times a \otimes x$$

and can be used to generate cofree comonoids .

A cofree functor is the right adjoint to the forgetful functor. Just like the left adjoint preserved coproducts, the right adjoint preserves products. One can therefore easily combine comonads using products (if the need arises to solve the coexpression problem).

Just like the monad is a monoid in the category of endofunctors, a comonad is a comonoid in the same category. The functor that generates a cofree comonad has the form:

```
type ComonadF f g = Identity :*: Compose f g
```

where the product of functors is defined as:

```
data (f :*: g) e = Both (f e) (g e)
infixr 6 :*:
```

Here's the more familiar form of this functor:

```
data ComonadF f g e = e :< f (g e)
```

It is indeed a higher order functor, as witnessed by this instance:

```
instance Functor f => HFunctor (ComonadF f) where
  hfmap nat (e :< fge) = e :< fmap nat fge
  ffmap h (e :< fge) = h e :< fmap (fmap h) fge
```

A cofree comonad is the terminal coalgebra for this functor and can be written as a fixed point:

```
type Cofree f = FixH (ComonadF f)
```

Indeed, for any functor `f`, `Cofree f` is a comonad:

```
instance Functor f => Comonad (Cofree f) where
  extract (InH (e :< fge)) = e
  duplicate fr@(InH (e :< fge)) =
               InH (fr :< fmap duplicate fge)
```

Cofree Comonad Example

The canonical example of a cofree comonad is an infinite stream:

```
type Stream = Cofree Identity
```

We can use this stream to sample a function. We'll encapsulate this function inside the following functor (in fact, itself a comonad):

```
data Store a x = Store a (a -> x)
    deriving Functor
```

We can use a higher order coalgebra to unpack the `Store` into a stream:

```
streamCoa :: HCoalgebra (ComonadF Identity)(Store Int)
streamCoa (Store n f) =
    f n :< (Identity $ Store (n + 1) f)
```

The actual unpacking is a higher order anamorphism:

```
stream :: Store Int a -> Stream a
stream = hana streamCoa
```

We can use it, for instance, to generate a list of squares of natural numbers:

```
stream (Store 0 (^2))
```

Since, in Haskell, the same fixed point defines a terminal coalgebra as well as an initial algebra, we are free to construct algebras and catamorphisms for streams. Here's an algebra that converts a stream to an infinite list:

```
listAlg :: HAlgebra (ComonadF Identity) []
listAlg(a :< Identity as) = a : as

toList :: Stream a -> [a]
toList = hcata listAlg
```

Future Directions

In this paper I concentrated on one type of higher order functor:

$$1 + a \otimes x$$

and its dual. This would be equivalent to studying folds for lists and unfolds for streams. But the structure of the functor category is richer than that. Just like basic data types can be combined into algebraic data types, so can functors. Moreover, besides the usual sums and products, the functor category admits at least two additional monoidal structures generated by functor composition and Day convolution.

Another potentially fruitful area of exploration is the profunctor category, which is also equipped with two monoidal structures, one defined by profunctor composition, and another by Day convolution. A free monoid with respect to profunctor composition is the basis of Haskell `Arrow` library [jaskelioff]. Profunctors also play an important role in the Haskell lens library [kmett].

Bibliography

1. Erik Meijer, Maarten Fokkinga, and Ross Paterson, Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire
2. Conor McBride, Ross Paterson, Idioms: applicative programming with effects
3. Paolo Capriotti, Ambrus Kaposi, Free Applicative Functors
4. Wouter Swierstra, Data types a la carte
5. Exequiel Rivas and Mauro Jaskelioff, Notions of Computation as Monoids
6. Edward Kmett, Lenses, Folds and Traversals
7. Richard Bird and Lambert Meertens, Nested Datatypes
8. Patricia Johann and Neil Ghani, Initial Algebra Semantics is Enough!

Next Page »