A Neighborhood of Infinity

Sunday, February 25, 2007

Monads for vector spaces, probability and quantum mechanics pt. I

I've been enjoying Eric Kidd's articles on probability theory with Haskell. So I thought I'd follow them up with two things: (1) finding the underlying algebraic structure and (2) showing how it's general enough to do more than just probability.

Firstly, Eric found a really neat factoring of the probability monad as a 'product' of two monads: the `Perhaps` monad and the `List` monad. This factoring can be interpreted algebraically.

A monoid is defined to be a set m with a binary operator (typically written as abuttal, ie. the product of a and b is ab) that is associative, and with an element, 1, that is an identity for the binary operator. The `List` monad gives rise to the monoid freely generated by a set, in other words it defines the smallest monoid containing the set and with no extra relationships between the elements that don't come from the definition of a monoid. The binary operator for `List` is called `++` and we embed the original set in the monoid using the function `\x -> [x]`, otherwise known as `return`. `[]` is the identity. It's not hard to see that we have asociativity as `(a ++ b) ++ c == a ++ (b ++ c)`. If we use `Set` instead of `List` then we get the free *commutative* monoid instead, ie. one where a+b=b+a.

(Eh? Where's my paragraph on m-sets. Don't tell me I have to format all those HTML subscripts again! I hate it when computers do that. I wrote a paragraph and it just vanished. Oh well, maybe version 2.0 will be better. But without subscripts this time...)

If m is a monoid then an m-set is a set s with an action of m on it. An action of a monoid on a set is a scheme that converts each element of the monoid, say g, into a map g^*:S→S so that 1^* is the identity map and $g^*h^*=(gh)^*$. The type `Writer m s` corresponds to pairs in s×m but this also doubles as the free m-set. We simply define $f^*(x,g)=(x,fg)$. It's free because we never get $f^*(x,f')=g^*(x',g')$ unless x=x', so we never get any surprising equalities that aren't inherent in the definition of an m-set. `Perhaps`, which is actually a pseudonym for `Writer Float`, defines an **R**-set, where **R** is the monoid of reals under multiplication.

In an earlier post I showed how you could layer up algebraic structures. There I showed that you could combine a commutative monoid structure with a monoid structure to get a semiring. So what happens if we layer up a commutative monoid and an m-set? We'll get something that has a commutative binary operator but that can also be acted on my elements of m. Specialise to the case when m is a field like the real or complex numbers. Does this sound like a familiar structure? I hope so, it's a description of a vector space. What Eric Kidd has shown is that we can build a vector space monad by applying an m-set monad transformer (with m a field) to a commutative monoid monad. (This isn't a completely trivial fact and it depends on the fact that the definition of `PerhapsT` handles distributivity correctly.) I think that's pretty neat. But I'm getting slightly ahead of myself here as I haven't shown what Eric's stuff has to do with vector spaces.

A probability distribution can be thought of as a vector with outcomes forming a basis. Any distribution attaches a probability 'weight' to each possible outcome in the same way that a vector can be written as a weighted sum of basis elements. Each time we have a probabilistic transition, we're effectively multiplying our distribution vector by a stochastic matrix. Eric (and others)'s monad allows you to write these matrix multiplications in a very natural way.

A vector space forms a monad in a straightforward way. If V(B) is the vector space generated by basis B, and V(C) is another vector space, then any function B→V(C) can be lifted to a linear map V(B)→V(C). This lift is clearly of type (B→V(C))→(V(B)→V(C)). With a little flip we get V(B)→(B→V(C))→V(C). This is the main step in showing V to be a monad. And what the probability monad allows us to do is write our stochastic matrices in terms of what happens to the individual basis elements (ie. outcomes) instead of having to write out the entire matrix.

Anyway, this is all just waffle and it really needs some code to make it more concrete. I have an ulterior motive here. It's not just probability theory that is described by vector spaces. So is quantum mechanics. And I plan to work my way up to defining quantum computers and implementing a bunch of well known quantum algorithms in Haskell. In the next installment I expect to get at least as far as writing the code to play with the square root of NOT operator.

PS This idea of layering up algebraic structures is one I haven't found in the textbooks yet. I'm guessing it's in Mac Lane, but I haven't yet justified the cost of that book to myself. But I don't actually have a clue what a category theorist would call a monad transformer and don't recall reading about them in any texts that weren't specifically about computer science. Maybe someone else can fill me in. I do know that it's absolutely standard to get free algebraic structures from monads.

Errata: As notfancy points out, I should really be talking about multisets rather than sets, otherwise we have a+a=a. As programmers often use lists to represent both sets and multisets it can sometimes get confusing.

Posted by Dan Piponi at Sunday, February 25, 2007

Labels: monad, physics

9 comments:

Eric Kidd said...

Great article! I'm looking forward to part 2.

As far as I know, Haskell's monad transformers are more commonly known as "monad morphisms" in category theory texts. A Google search turns up a lot of CS-flavored papers.

Sunday, 25 February, 2007

alpheccar said...

There are lots of links between monads and algebras and for each monad you can build a free algebra.

You can find more in this paper

Sunday, 25 February, 2007

Michi said...

The one place I really found layered algebraic structures of the kind you're talking about in a larger extent than the trivial recognitions (as in "Uh, yeah, a ring is an abelian group and a monoid") would be with enriched categories.

Basically, what you do, is to give your hom-sets additional algebraic structure, and show that this additional structure says a lot more about the category. So any additive category is really an Ab-enriched category.

Monday, 26 February, 2007

sigfpe said...

michi,

"Uh, yeah, a ring is an abelian group and a monoid"

I have to admit, I started worrying about the triviality of all this when you said that. But then I remembered that there is something neat going on because it's not just about recognition. We have a menu of algebraic parts (monoid, m-set, etc.) and a bunch of Haskell functions that build algebraic structure from these parts correctly and with all of the requisite laws (eg. distributivity) obeyed.

Anyway, you're the third person in about a many weeks to tell me to read up on enriched categories. Unfortunately I need to swot up more on ordinary categories first.

Monday, 26 February, 2007

Michael said...

Re: eric kidd.

It seems to me that Haskell's monad transformers are different but related to monad morphisms in the following ways:

Per Mac Lane pg. 142, Exercise 3(a), monads <T, N, X> over a category P form a category with objects all the monads over P and with arrows given by natural tranformations a: T -*> T' satisfying appropriate conditions.

Now since we have a category, every monad induces an identity (monad) morphism, including the monads which cannot be "transformerized" (i.e. which cannot be wrapped around arbitrary inner monads).

Second, monad transformers, being polymorphic in the monad they wrap, appear to me to denote entire families of monad morphisms. Sometimes, when a monad transformer can wrap an arbitrary monad, this family of morphisms is "large" in the sense that every monad is the source of some morphism in the family. Other times, such as for the List monad transformer [1], the family is somewhat smaller.

Third, there's actually a variety of categories (truthfully, 2-categories) which have monads as their objects/0-cells. Leinster[2] concisely describes four such "2-categories of monads", more specifically, those with lax, colax, weak, and strict maps as their 1-cells. These formulations of "monad morphism" are particularly interesting because they allow one to change the category over which the monad is built. This possibility creates two interesting new questions concerning the meaning of computation which I have never seen addressed:

1) An inverse problem: what interesting (category, monad) pairs can be mapped *into* a monad over "Hask"?

2) What can monads over "Hask" be mapped to in other interesting categories?

The former question feels like abstract interpretation to me: we have a more detailed model of what's happening that we're squishing to fit into Haskell's monads, types, etc. Alternately, we might have some really "simple" model and we want to describe a canonical embedding into Haskell.

The latter question feels like an abstraction question: for example, can you concisely express the halting problem as a map from (Hask, IO) to the "Halting category", where the only arrows between types are "Bottom", which diverges and "Top", which halts?

Either way, these comparisons show that Haskell's monad transformers are related to but not the same as the monad morphisms in one particular category of monads, that there's lots of unexplored territory figuring out the relationship between Haskell's monads and the monads of other categories, and that the various 2-categories of monads appear offer an adequately precise language for conducting the latter exploration.

Notes:

[1] Mark P. Jones states, in his "Monad Transformers and Modular Interpreters" that the List monad can only wrap commutative monads.

[2] See section 6.1 of his book "Higher Operads, Higher Categories", which is available as math.CT/0305049 on arXiv.

Monday, 26 February, 2007

Chung-chieh Shan said...

Indeed, monad transformers are not monad morphisms. The former are monad constructions whereas the latter are equivalent to monad layerings.

Tuesday, 27 February, 2007

Miles said...

Well, I can't help too much, because I don't (yet) understand monad transformers. My working hypothesis is that they have something to do with distributive laws. There are various ways of describing distributive laws (including as a certain type of monad morphism), but the basic idea is that if you have monads S and T, a distributive law of S over T is the extra data you need to give a monad structure to ST. The right choice of distributive law of monoids over commutative monoids gives you the theory of rigs (aka semirings), whence the name. This isn't the only way of combining monads: if your monads are finitary, there's also a notion of their tensor product. The tensor product of the "free monoid" monad with itself is the "free commutative monoid" monad, by the Eckmann-Hilton lemma.

Enriched categories are powerful stuff, and I should really learn more about them. But the basic idea is simple: take some monoidal category E (ie, there's a functor *:E^2 -> E and a unit u in E, and (E, *, u) behaves like a monoid up to isomorphism). A category C enriched in E has hom-E-objects instead of hom-sets between any two objects of C. Composition is a family of E-maps o:C(B,C)*C(A,B) -> C(A,C), satisfying the usual diagrams. Enriching in Ab gives an abelian group structure on your hom-sets, so you can add morphisms; enriching in Top gives a topological structure to your hom-sets. But there's no reason your E-objects need to be sets or anything like them: for instance, categories enriched in (R,+,0) are something very close to metric spaces.

There's a rich and detailed theory of enriched categories, but the short version is apparently "if E is symmetric monoidal closed, then all of ordinary category theory carries over, mutas mutandis". You have to re-jig some definitions so they make sense, but pretty much everything works.

Wednesday, 04 April, 2007

Philip Wadler said...

You can find constructs similar to the one you describe in my paper with David King, Combining Monads.

Sunday, 04 October, 2009

Anonymous said...

I have also used WriterT to make probability distributions. Except that, I generalized the probabilities to be any normalizable semiring.

Friday, 11 November, 2011

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

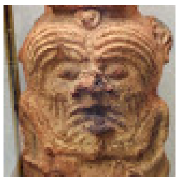
Blog Archive

- ► 2014 (4)
- ► 2013 (4)
- ► 2012 (8)
- ► 2011 (13)
- ► 2010 (20)
- ► 2009 (21)
- ► 2008 (35)
- ▼ 2007 (37)
  - ► December (1)
  - ► November (5)
  - ► October (2)
  - ► September (3)
  - ► July (3)
  - ► June (2)
  - ► May (1)
  - ► April (3)
  - ► March (7)
  - ▼ February (6)
    - Monads in C, pt. II
    - Monads for vector spaces, probability and quantum ...
    - The Essence of Quantum Computing
    - Modular arithmetic with regular expressions
    - Exceptions, Disjunctions and Continuations
    - Comonads and reading from the future
  - ► January (4)
- ► 2006 (92)
- ► 2005 (53)

Some Links

- The Comonad.Reader
- Rubrication
- Richard Borcherds: Mathematics and physics
- The n-Category Cafe
- Ars Mathematica

About Me



Dan Piponi

Follow 2.5k

Blog: A Neighborhood of Infinity
Code: Github
Twitter: sigfpe
Home page: www.sigfpe.com

View my complete profile