

Category theory and functional programming

Mikael Vejdemo-Johansson

Thursday 2nd February, 2012

Contents

Contents	2
1 First definitions	5
1.1 Introduction	5
1.2 Category	7
2 Special morphisms, special objects	17
3 Functors	27
4 Products, coproducts, dualization	35
5 Limits and colimits	41
6 Pairs of constructions	51
7 Monoid objects and monads	61
8 Algebras	69
9 The morphism zoo	79
10 Topoi	81

Introduction

Introduction to the book.

First definitions

1.1 Introduction

Why this course?

An introduction to Haskell will usually come with pointers toward Category Theory as a useful tool, though not with much more than the mention of the subject. This course is intended to fill that gap, and provide an introduction to Category Theory that ties into Haskell and functional programming as a source of examples and applications.

What will we cover?

The definition of categories, special objects and morphisms, functors, natural transformation, (co-)limits and special cases of these, adjunctions, freeness and presentations as categorical constructs, monads and Kleisli arrows, recursion with categorical constructs.

Maybe, just maybe, if we have enough time, we'll finish with looking at the definition of a topos, and how this encodes logic internal to a category. Applications to fuzzy sets.

What do we require?

Our examples will be drawn from discrete mathematics, logic, Haskell programming and linear algebra. I expect the following concepts to be at least vaguely familiar to anyone taking this course:

- Sets
- Functions
- Permutations
- Groups
- Partially ordered sets

1. First definitions

- Vector spaces
- Linear maps
- Matrices
- Homomorphisms

Good references

On reserve in the mathematics/CS library are:

Mac Lane: Categories for the working mathematician This book is technical, written for a mathematical audience, and puts in more work than is strictly necessary in many of the definitions. When Awodey and Mac Lane deviate, we will give Awodey priority.

Awodey: Category Theory This book is also available as an ebook, accessible from the Stanford campus network. The course-work webpage has links to the ebook under Materials.

Monoids

In order to settle notation and ensure everybody's seen a definition before:

Definition A *monoid* is a set M equipped with a binary associative operation $*$ (in Haskell: `mappend`) and an identity element \emptyset (in Haskell: `mempty`).

A *semigroup* is a monoid without the requirement for an identity element.

A function $f : M \rightarrow N$ is a monoid homomorphism if the following conditions hold:

- $f(\emptyset) = \emptyset$
- $f(m * m') = f(m) * f(m')$

Examples

- Any group is a monoid. Thus, specifically, the integers with addition is a monoid.
- The natural numbers, with addition.
- Strings L^* in an alphabet L is a monoid with string concatenation forming the operation and the empty string the identity.
- Non-empty strings form a semigroup.

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

For more information, see the Wikipedia page on Monoids. <http://en.wikipedia.org/wiki/Monoid>.

Awodey: p. 10.

Partially ordered set

Definition A *partially ordered set*, or a *partial order*, or a *poset* is a set P equipped with a binary relation \leq which is:

- Reflexive: $x \leq x$ for all $x \in P$
- Anti-symmetric: $x \leq y$ and $y \leq x$ implies $x = y$ for all $x, y \in P$.
- Transitive: $x \leq y$ and $y \leq z$ implies $x \leq z$ for all $x, y, z \in P$.

If $x \leq y$ or $y \leq x$, we call x, y *comparable*. Otherwise we call them *incomparable*. A poset where all elements are mutually comparable is called a *totally ordered set* or a *total order*.

If we drop the requirement for anti-symmetry, we get a *pre-ordered set* or a *pre-order*.

If we have several posets, we may indicate which poset we're comparing *in* by indicating the poset as a subscript to the relation symbol.

A *monotonic* map of posets is a function $f : P \rightarrow Q$ such that $x \leq_P y$ implies $f(x) \leq_Q f(y)$.

Examples

- The reals, natural numbers, integers all are posets with the usual comparison relation. A poset in which all elements are comparable.
- The natural numbers, excluding 0, form a poset with $a \leq b$ if $a|b$.
- Any family of subsets of a given set form a poset with the order given by inclusion.

For more information, see the Wikipedia page on Partially ordered set. http://en.wikipedia.org/wiki/Partially_ordered_set

Awodey: p. 6. Preorders are defined on page 8-9.

1.2 Category

Awodey has a slightly different exposition. Relevant pages in Awodey for this lecture are: sections 1.1-1.4 (except Def. 1.2), 1.6-1.8.

Graphs

We recall the definition of a (*directed*) *graph*. A graph G is a collection of *edges* (*arrows*) and *vertices* (*nodes*). Each edge is assigned a *source* node and a *target* node.

$source \rightarrow target$

Given a graph G , we denote the collection of nodes by G_0 and the collection of arrows by G_1 . These two collections are connected, and the graph given its structure, by two functions: the source function $s : G_1 \rightarrow G_0$ and the target function $t : G_1 \rightarrow G_0$.

We shall not, in general, require either of the collections to be a set, but will happily accept larger collections; dealing with set-theoretical paradoxes as and when we have to. A graph where both nodes and arrows are sets shall be called *small*. A graph where either is a class shall be called *large*.

If both G_0 and G_1 are finite, the graph is called *finite* too.

The *empty graph* has $G_0 = G_1 = \emptyset$.

A *discrete graph* has $G_1 = \emptyset$.

A *complete graph* has $G_1 = \{(v, w) | v, w \in G_0\}$.

A *simple graph* has at most one arrow between each pair of nodes. Any relation on a set can be interpreted as a simple graph.

Show some examples.

A *homomorphism* $f : G \rightarrow H$ of graphs is a pair of functions $f_0 : G_0 \rightarrow H_0$ and $f_1 : G_1 \rightarrow H_1$ such that sources map to sources and targets map to targets, or in other words:

$$\cdot \quad s(f_1(e)) = f_0(s(e))$$

$$\cdot \quad t(f_1(e)) = f_0(t(e))$$

By a *path* in a graph G from the node x to the node y of length k , we mean a sequence of edges (f_1, f_2, \dots, f_k) such that:

$$\cdot \quad s(f_1) = x$$

$$\cdot \quad t(f_k) = y$$

$$\cdot \quad s(f_i) = t(f_{i-1}) \text{ for all other } i.$$

Paths with start and end point identical are called *closed*. For any node x , there is a unique closed path $()$ starting and ending in x of length 0.

For any edge f , there is a unique path from $s(f)$ to $t(f)$ of length 1: (f) .

We denote by G_k the set of paths in G of length k .

DRAFT: Thursday 2nd February, 2012 16:49

Categories

We now are ready to define a category. A *category* is a graph C equipped with an associative composition operation $\circ : G_2 \rightarrow G_1$, and an identity element for composition 1_x for each node x of the graph.

Note that G_2 can be viewed as a subset of $G_1 \times G_1$, the set of all pairs of arrows. It is intentional that we define the composition operator on only a subset of the set of all pairs of arrows - the composable pairs. Whenever you'd want to compose two arrows that don't line up to a path, you'll get nonsense, and so any statement about the composition operator has an implicit "whenever defined" attached to it.

The definition is not quite done yet - this composition operator, and the identity arrows both have a few rules to fulfill, and before I state these rules, there are some notation we need to cover.

Backwards!

If we have a path given by the arrows (f, g) in G_2 , we expect $f : A \rightarrow B$ and $g : B \rightarrow C$ to compose to something that goes $A \rightarrow C$. The origin of all these ideas lies in geometry and algebra, and so the abstract arrows in a category are *supposed* to behave like functions under function composition, even though we don't say it explicitly.

Now, we are used to writing function application as $f(x)$ - and possibly, from Haskell, as $f \ x$. This way, the composition of two functions would read $g(f(x))$.

On the other hand, the way we write our paths, we'd read f then g . This juxtaposition makes one of the two ways we write things seem backwards. We can resolve it either by making our paths in the category go backwards, or by reversing how we write function application.

In the latter case, we'd write $x.f$, say, for the application of f to x , and then write $x.f.g$ for the composition. It all ends up looking a lot like Reverse Polish Notation, and has its strengths, but feels unnatural to most. It does, however, have the benefit that we can write out function composition as $(f, g) \mapsto f.g$ and have everything still make sense in all notations.

In the former case, which is the most common in the field, we accept that paths as we read along the arrows and compositions look backwards, and so, if $f : A \rightarrow B$ and $g : B \rightarrow C$, we write $g \circ f : A \rightarrow C$, remembering that *elements* are introduced from the right, and the functions have to consume the elements in the right order.

--- ---

The existence of the identity map can be captured in a function language as well: it is the existence of a function $u : G_0 \rightarrow G_1$.

Now for the remaining rules for composition. Whenever defined, we expect associativity - so that $h \circ (g \circ f) = (h \circ g) \circ f$. Furthermore, we expect:

1. Composition respects sources and targets, so that:

$$\begin{aligned} \cdot \quad s(g \circ f) &= s(f) \\ \cdot \quad t(g \circ f) &= t(g) \end{aligned}$$

2. $s(u(x)) = t(u(x)) = x$

In a category, arrows are also called *morphisms*, and nodes are also called *objects*. This ties in with the algebraic roots of the field.

We denote by $\text{Hom}_C(A, B)$, or if C is obvious from context, just $\text{Hom}(A, B)$, the set of all arrows from A to B . This is the *hom-set* or *set of morphisms*, and may also be denoted $C(A, B)$.

If a category is large or small or finite as a graph, it is called a large/small/finite category.

A category with objects a collection of sets and morphisms a selection from all possible set-valued functions such that the identity morphism for each object is a morphism, and composition in the category is just composition of functions is called *concrete*. Concrete categories form a very rich source of examples, though far from all categories are concrete.

Again, the Wikipedia page on Category (mathematics) http://en.wikipedia.org/wiki/Category_%28mathematics\%29 is a good starting point for many things we will be looking at throughout this course.

New Categories from old

As with most other algebraic objects, one essential part of our tool box is to take known objects and form new examples from them. This allows us generate a wealth of examples from the ones that shape our intuition.

Typical things to do here would be to talk about *subobjects*, *products* and *coproducts*, sometimes obvious *variations on the structure*, and what a *typical object* looks like. Remember from linear algebra how *subspaces*, *cartesian products* (which for finite-dimensional vectorspaces covers both products and coproducts) and *dual spaces* show up early, as well as the theorems giving *dimension* as a complete descriptor of a vector space.

DRAFT: Thursday 2nd February, 2012 16:49

We'll go through the same sequence here; with some significant small variations.

A category D is a *subcategory* of the category C if:

- $D_0 \subseteq C_0$
- $D_1 \subseteq C_1$
- D_1 contains 1_X for all $X \in D_0$
- sources and targets of all the arrows in D_1 are all in D_0
- the composition in D is the restriction of the composition in C .

Written this way, it does look somewhat obnoxious. It does become easier though, with the realization - studied closer in homework exercise 2 - that the really important part of a category is the collection of arrows. Thus, a subcategory is a subcollection of the collection of arrows - with identities for all objects present, and with at least all objects that the existing arrows imply.

A subcategory $D \subseteq C$ is *full* if $D(A, B) = C(A, B)$ for all objects A, B of D . In other words, a full subcategory is completely determined by the selection of objects in the subcategory.

A subcategory $D \subseteq C$ is *wide* if the collection of objects is the same in both categories. Hence, a wide subcategory picks out a subcollection of the morphisms.

The *dual* of a category is to a large extent inspired by vector space duals. In the dual C^* of a category C , we have the same objects, and the morphisms are given by the equality $C^*(A, B) = C(B, A)$ - every morphism from C is present, but it goes in the *wrong* direction. Dualizing has a tendency to add the prefix *co*- when it happens, so for instance coproducts are the dual notion to products. We'll return to this construction many times in the course.

Given two categories C, D , we can combine them in several ways:

1. We can form the category that has as objects the disjoint union of all the objects of C and D , and that sets $Hom(A, B) = \emptyset$ whenever A, B come from different original categories. If A, B come from the same original category, we simply take over the homset from that category. This yields a categorical *coproduct*, and we denote the result by $C + D$. Composition is inherited from the original categories.
2. We can also form the category with objects $\langle A, B \rangle$ for every pair of objects $A \in C, B \in D$. A morphism in $Hom(\langle A, B \rangle, \langle A', B' \rangle)$ is simply a pair $\langle f : A \rightarrow A', g : B \rightarrow B' \rangle$. Composition is

defined componentwise. This category is the categorical correspondent to the cartesian *product*, and we denote it by $C \times D$.

In these three constructions - the dual, the product and the co-product - the arrows in the categories are formal constructions, not functions; even if the original category was given by functions, the result is no longer given by a function.

Given a category C and an object A of that category, we can form the *slice category* C/A . Objects in the slice category are arrows $B \rightarrow A$ for some object B in C , and an arrow $\phi : f \rightarrow g$ is an arrow $s(f) \rightarrow s(g)$ such that $f = g \circ \phi$. Composites of arrows are just the composites in the base category.

Notice that the same arrow ϕ in the base category C represents potentially many different arrows in C/A : it represents one arrow for each choice of source and target compatible with it.

There is a dual notion: the *coslice category* $A \backslash C$, where the objects are paired with maps $A \rightarrow B$.

Slice categories can be used, among other things, to specify the idea of parametrization. The slice category C/A gives a sense to the idea of *objects from C labeled by elements of A* .

We get this characterization by interpreting the arrow representing an object as representing its source and a *type function*. Hence, in a way, the `Typeable` type class in Haskell builds a slice category on an appropriate subcategory of the category of datatypes.

Alternatively, we can phrase the importance of the arrow in a slice categories of, say, `Set`, by looking at preimages of the slice functions. That way, an object $f : B \rightarrow A$ gives us a family of (disjoint) subsets of B *indexed* by the elements of A .

Finally, any graph yields a category by just filling in the arrows that are missing. The result is called the *free category generated by the graph*, and is a concept we will return to in some depth. Free objects have a strict categorical definition, and they serve to give a model of thought for the things they are free objects for. Thus, categories are essentially graphs, possibly with restrictions or relations imposed; and monoids are essentially strings in some alphabet, with restrictions or relations.

Examples

- The empty category.
 - No objects, no morphisms.
- The one object/one arrow category 1.

DRAFT: Thursday 2nd February, 2012 16:49

- A single object and its identity arrow.
- The categories 2 and $1 + 1$.
 - Two objects, A, B with identity arrows and a unique arrow $A \rightarrow B$.
- The category **Set** of sets.
 - Sets for objects, functions for arrows.
- The category **FSet** of finite sets.
 - Finite sets for objects, functions for arrows.
- The category **PFn** of sets and partial functions.
 - Sets for objects. Arrows are pairs $(S' \subseteq S, f : S' \rightarrow T) \in PFn(S, T)$.
 - $PFn(A, B)$ is a partially ordered set. $(S_f, f) \leq (S_g, g)$ precisely if $S_f \subseteq S_g$ and $f = g|_{S_f}$.
 - The exposition at Wikipedia uses the construction here:
http://en.wikipedia.org/wiki/Partial_function.
- There is an alternative way to define a category of partial functions: For objects, we take sets, and for morphisms $S \rightarrow T$, we take subsets $F \subseteq S \times T$ such that each element in S occurs in at most one pair in the subset. Composition is by an interpretation of these subsets corresponding to the previous description. We'll call this category PFn' .
- Every partial order is a category. Each hom-set has at most one element.
 - Objects are the elements of the poset. Arrows are unique, with $A \rightarrow B$ precisely if $A \leq B$.
- Every monoid is a category. Only one object. The elements of the monoid correspond to the endo-arrows of the one object.
- The category of **Sets** and injective functions.
 - Objects are sets. Morphisms are injective functions between the sets.
- The category of **Sets** and surjective functions.
 - Objects are sets. Morphisms are surjective functions between the sets.

1. First definitions

- The category of k -vector spaces and linear maps.
- The category with objects the natural numbers and $Hom(m, n)$ the set of $m \times n$ -matrices.
 - Composition is given by matrix multiplication.
- The category of Data Types with Computable Functions.
 - Our ideal programming language has:
 - * Primitive data types.
 - * Constants of each primitive type.
 - * Operations, given as functions between types.
 - * Constructors, producing elements from data types, and producing derived data types and operations.
 - We will assume that the language is equipped with
 - * A do-nothing operation for each data type. Haskell has `id`.
 - * An empty type `1`, with the property that each type has exactly one function to this type. Haskell has `()`. We will use this to define the constants of type t as functions $1 \rightarrow t$. Thus, constants end up being 0-ary functions.
 - * A composition constructor, taking an operator $f : A \rightarrow B$ and another operator $g : B \rightarrow C$ and producing an operator $g \circ f : A \rightarrow C$. Haskell has `(.)`.
 - This allows us to model a functional programming language with a category.
- The category with objects logical propositions and arrows proofs.
- The category `Rel` has objects finite sets and morphisms $A \rightarrow B$ being subsets of $A \times B$. Composition is by $(a, c) \in g \circ f$ if there is some $b \in B$ such that $(a, b) \in f, (b, c) \in g$. Identity morphism is the diagonal $(a, a) : a \in A$.

Homework

For a passing mark, a written, acceptable solution to at least 3 of the 6 questions should be given no later than midnight before the next lecture.

For each lecture, there will be a few exercises marked with the symbol `*`. These will be more difficult than the other exercises given, will require significant time and independent study, and will aim to complement the course with material not covered in

DRAFT: Thursday 2nd February, 2012 16:49

lectures, but nevertheless interesting for the general philosophy of the lecture course.

1. Prove the general associative law: that for any path, and any bracketing of that path, the same composition results.
2. Which of the following form categories? Proof and disproof for each:
 - a) Objects are finite sets, morphisms are functions such that $|f^{-1}(b)| \leq 2$ for all morphisms f , objects B and elements b .
 - b) Objects are finite sets, morphisms are functions such that $|f^{-1}(b)| \geq 2$ for all morphisms f , objects B and elements b .
 - c) Objects are finite sets, morphisms are functions such that $|f^{-1}(b)| < \infty$ for all morphisms f , objects B and elements b .

Recall that $f^{-1}(b) = \{a \in A : f(a) = b\}$.

3. Suppose $u : A \rightarrow A$ in some category C .
 - a) If $g \circ u = g$ for all $g : A \rightarrow B$ in the category, then $u = 1_A$.
 - b) If $u \circ h = h$ for all $h : B \rightarrow A$ in the category, then $u = 1_A$.
 - c) These two results characterize the objects in a category by the properties of their corresponding identity arrows completely. Specifically, there is a way to rephrase the definition of a category such that everything is stated in terms of arrows.
4. For as many of the examples given as you can, prove that they really do form a category. Passing mark is at least 60% of the given examples.
5. Which of the categories are subcategories of which other categories? Which of these are wide? Which are full?
6. For this question, all parts are required:
 - a) For which sets is the free monoid on that set commutative.
 - b) Prove that for any category C , the set $\text{Hom}(A, A)$ is a monoid under composition for every object A .
For details on the construction of a free monoid, see the Wikipedia pages on the Free Monoid http://en.wikipedia.org/wiki/Free_monoid and on the Kleene star http://en.wikipedia.org/wiki/Kleene_star.

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

7. * Read up on ω -complete partial orders. Suppose S is some set and \mathfrak{P} is the set of partial functions $S \rightarrow S$ - in other words, an element of \mathfrak{P} is some pair $(S_0, f : S_0 \rightarrow S)$ with $S_0 \subseteq S$. We give this set a poset structure by $(S_0, f) \leq (S_1, g)$ precisely if $S_0 \subseteq S_1$ and $f(s) = g(s) \forall s \in S_0$.

- a) Show that \mathfrak{P} is a strict ω -CPO.
- b) An element x of S is a *fixpoint* of $f : S \rightarrow S$ if $f(x) = x$. Let \mathfrak{N} be the ω -CPO of partially defined functions on the natural numbers. We define a function $\phi : \mathfrak{N} \rightarrow \mathfrak{N}$ by sending some $h : \mathbb{N} \rightarrow \mathbb{N}$ to a function k defined by
 - i. $k(0) = 1$
 - ii. $k(n)$ is defined only if $h(n-1)$ is defined, and then by $k(n) = n * h(n-1)$.

Describe $\phi(n \mapsto n^2)$ and $\phi(n \mapsto n^3)$. Show that ϕ is *continuous*. Find a fixpoint (S_0, f) of ϕ such that any other fixpoint of the same function is larger than this one.

Find a continuous endofunction on some ω -CPO that has the fibonacci function $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$ as the least fixed point.

Implement a Haskell function that finds fixed points in an ω -CPO. Implement the two fixed points above as Haskell functions - using the ω -CPO fixed point approach in the implementation. It may well be worth looking at `Data.Map` to provide a Haskell context for a partial function for this part of the task.

Lecture 2

Special morphisms, special objects

This lecture covers material occurring in Awodey sections 2.1-2.4.

Morphisms and objects

Some morphisms and some objects are special enough to garner special names that we will use regularly.

In morphisms, the important properties are

- cancellability - the categorical notion corresponding to properties we use when solving, e.g., equations over \mathbb{N} :

$$3x = 3y \Rightarrow x = y$$

See also Wikipedia http://en.wikipedia.org/wiki/Cancellation_property, where the relevant definitions and some interesting keywords occur. The article is technical and terse, though.

- existence of inverses - which is stronger than cancellability. If there are inverses around, this implies cancellability, by applying the inverse to remove the common factor. Cancellability, however, does not imply that inverses exist: we can cancel the 3 above, but this does not imply the existence of $1/3 \in \mathbb{N}$.

Thus, we'll talk about isomorphisms - which have two-sided inverses, monomorphisms and epimorphisms - which have cancellability properties, and split morphisms - which are mono's and epi's with corresponding one-sided inverses. We'll talk about what these concepts - defined in terms of equationsolving with arrows - apply to more familiar situations. And we'll talk about how the semantics of some of the more wellknown ideas in mathematics are captured by these notions.

For objects, the properties are interesting in what happens to homsets with the special object as source or target. An empty homset is pretty boring, and a *large* homset is pretty boring. The real power, we find, is when *all* homsets with the specific source

or target are singleton sets. This allows us to formulate the idea of a 0 in categorical terms, as well as capturing the roles of the empty set and of elements of sets - all using only arrows.

Isomorphisms

An arrow $f : A \rightarrow B$ in a category C is an *isomorphism* if it has a twosided inverse g . In other words, we require the existence of a $g : B \rightarrow A$ such that $fg = 1_B$ and $gf = 1_A$.

In Set

In a category of sets with structure with morphisms given by functions that respect the set structure, isomorphism are bijections respecting the structure. In the category of sets, the isomorphisms are bijections.

In wikipedia: <http://en.wikipedia.org/wiki/Bijection>

Representative subcategories

Very many mathematical properties and invariants are interesting because they hold for objects regardless of how, exactly, the object is built. As an example, most set theoretical properties are concerned with how large the set is, but not what the elements really are.

If all we care about are our objects up to isomorphisms, and how they relate to each other - we might as well restrict ourselves to one object for each isomorphism class of objects.

Doing this, we get a *representative subcategory*: a subcategory such that every object of the supercategory is isomorphic to some object in the subcategory.

The representative subcategory ends up being a more categorically interesting concept than the idea of a wide subcategory: it doesn't hit every object in the category, but it hits every object worth hitting in order to capture all the structure.

Example The category of finite sets has a representative subcategory given by all sets $[n] = \{1, \dots, n\}$.

Groupoids

A *groupoid* is a category where *all* morphisms are isomorphisms. The name originates in that a groupoid with one object is a bona fide group; so that groupoids are the closest equivalent, in one sense, of groups as categories.

A very rich starting point is the wikipedia page <http://en.wikipedia.org/wiki/Groupoid>. In the categorical definition on

DRAFT: Thursday 2nd February, 2012 16:49

this page, the difference to a category is in the existence and properties of the function `inv`.

Monomorphisms

We say that an arrow f is *left cancellable* if for any arrows g_1, g_2 we can show $f g_1 = f g_2 \Rightarrow g_1 = g_2$. In other words, it is left cancellable, if we can remove it from the far left of any equation involving arrows.

We call a left cancellable arrow in a category a *monomorphism*.

In Set

Left cancellability means that if, when we do first g_1 and then f we get the same as when we do first g_2 and then f , then we had equality already before we followed with f .

In other words, when we work with functions on sets, f doesn't introduce relations that weren't already there. Anything non-equal before we apply f remains non-equal in the image. This, translated to formulae gives us the well-known form for *injectivity*:

$$\begin{aligned}x \neq y &\Rightarrow f(x) \neq f(y) \text{ or moving out the negations,} \\f(x) = f(y) &\Rightarrow x = y.\end{aligned}$$

In Wikipedia: http://en.wikipedia.org/wiki/Injective_function.

Subobjects

Consider the subset $\{1, 2\} \subset \{1, 2, 3\}$. This is the image of an accordingly chosen injective map from any 2-element set into $\{1, 2, 3\}$. Thus, if we want to translate the idea of a subset into categorical language, it is not enough talking about monomorphisms, though the fact that inclusion is an injection indicates that we are on the right track.

The trouble that remains is that we do not want to view $\{1, 2\}$ as different subsets when it occurs as an image of the 2-element set $\{1, 2\}$ or when it occurs as an image of the 2-element set $\{5, 6\}$. So we need some way of figuring out how to catch these situations and parry for them.

We'll say that a morphism f *factors through* a morphism g if there is some morphism h such that $f = gh$.

We can also talk about a morphism $f : A \rightarrow C$ factoring through an *object* B by requiring the existence of morphisms $g : A \rightarrow B, h : B \rightarrow C$ that compose to f .

Now, we can form an equivalence relation on monomorphisms into an object A , by saying $f \sim g$ if f factors through g and g factors through f . The arrows implied by the factoring are inverse

to each other, and the source objects of equivalent arrows are isomorphic.

Equipped with this equivalence relation, we define a *subobject* of an object A to be an equivalence class of monomorphisms.

Wikipedia has an accurate exposition <http://en.wikipedia.org/wiki/Subobject>.

Epimorphisms

Right cancellability, by duality, is the implication $:g_1 f = g_2 f \Rightarrow g_1 = g_2$. The name, here comes from that we can remove the right cancellable f from the right of any equation it is involved in.

A right cancellable arrow in a category is an *epimorphism*.

In Set

For epimorphisms the interpretation in set functions is that whatever f does, it doesn't hide any part of the things g_1 and g_2 do. So applying f first doesn't influence the total available scope g_1 and g_2 have.

In Wikipedia: http://en.wikipedia.org/wiki/Surjective_function.

More on factoring

In Set, and in many other categories, any morphism can be expressed by a factorization of the form $f = ip$ where i is a monomorphism and p is an epimorphism. For instance, in Set, we know that a function is surjective onto its image, which in turn is a subset of the domain, giving a factorization into an epimorphism - the projection onto the image - followed by a monomorphism - the inclusion of the image into the domain.

A generalization of this situation is sketched out on the Wikipedia page for Factorization systems http://en.wikipedia.org/wiki/Factorization_system.

Note that in Set, every morphism that is both a mono and an epi is immediately an isomorphism. We shall see in the homework that the converse does not necessarily hold.

Initial and Terminal objects

An object 0 is *initial* if for every other object C , there is a unique morphism $0 \rightarrow C$. Dually, an object 1 is *terminal* if there is a unique morphism $C \rightarrow 1$.

DRAFT: Thursday 2nd February, 2012 16:49

First off, we note that the uniqueness above makes initial and terminal objects unique up to isomorphism whenever they exist: we shall perform the proof for one of the cases, the other is almost identical.

Proposition Initial (terminal) objects are unique up to isomorphism.

Proof: Suppose C and C' are both initial (terminal). Then there is a unique arrow $C \rightarrow C'$ and a unique arrow $C' \rightarrow C$. The compositions of these arrows are all endoarrows of one or the other. Since *all* arrows from (to) an initial (terminal) objects are unique, these compositions have to be the identity arrows. Hence the arrows we found between the two objects are isomorphisms. QED.

- In Sets, the empty set is initial, and any singleton set is terminal.
- In the category of Vector spaces, the single element vector space 0 is both initial and terminal.

On Wikipedia, there is a terse definition, and a good range of examples and properties: http://en.wikipedia.org/wiki/Initial_and_terminal_objects.

Note that terminal objects are sometimes called *final*, and are as such used in the formal logic specification of algebraic structures.

Zero objects

This last example is worth taking up in higher detail. We call an object in a category a *zero object* if it is simultaneously initial and terminal.

Some categories exhibit a richness of structure similar to the category of vectorspaces: all kernels exist (nullspaces), homsets are themselves abelian groups (or even vectorspaces), et.c. With the correct amount of richness, the category is called an *Abelian category*, and forms the basis for *homological algebra*, where techniques from topology are introduced to study algebraic objects.

One of the core requirements for an Abelian category is the existence of zero objects in it: if a category does have a zero object 0, then for any $\text{Hom}(A, B)$, the composite $A \rightarrow 0 \rightarrow B$ is a uniquely determined member of the homset, and the addition on the homsets of an Abelian category has this particular morphism as its identity element.

Pointless sets and generalized elements

Arrows to initial objects and from terminal objects are interesting too - and as opposed to the arrows from initial and to the terminals, there is no guarantee for these arrows to be uniquely determined. Let us start with arrows $A \rightarrow 0$ into initial objects.

In the category of sets, such an arrow only exists if A is already the empty set.

In the category of all monoids, with monoid homomorphisms, we have a zero object, so such an arrow is uniquely determined.

For arrows $1 \rightarrow A$, however, the situation is significantly more interesting. Let us start with the situation in **Set**. 1 is some singleton set, hence a function from 1 picks out one element as its image. Thus, at least in **Set**, we get an isomorphism of sets $A = \text{Hom}(1, A)$.

As with so much else here, we build up a general definition by analogy to what we see happening in the category of sets. Thus, we shall say that a *global element*, or a *point*, or a *constant* of an object A in a category with terminal objects is a morphism $x : 1 \rightarrow A$.

This allows us to talk about elements without requiring our objects to even be sets to begin with, and thus reduces everything to a matter of just morphisms. This approach is fruitful both in topology and in Haskell, and is sometimes called *pointless*.

The important point here is that we can replace *function application* $f(x)$ by the already existing and studied *function composition*. If a constant x is just a morphism $x : 1 \rightarrow A$, then the value $f(x)$ is just the composition $f \circ x : 1 \rightarrow A \rightarrow B$. Note, also, that since 1 is terminal, it has exactly one point.

In the idealized Haskell category, we have the same phenomenon for constants, but slightly disguised: a global constant is 0-ary function. Thus the type declaration `x :: a` can be understood as syntactic sugar for the type declaration `x :: () -> a` thus reducing everything to function types.

Similarly to the *global elements*, it may be useful to talk about *variable elements*, by which we mean non-specified arrows $f : T \rightarrow A$. Allowing T to range over all objects, and f to range over all morphisms into A , we are able to recover some of the element-centered styles of arguments we are used to. We say that f is *parametrized over* T .

Using this, it turns out that f is a monomorphism if for any variable elements $x, y : T \rightarrow A$, if $x \neq y$ then $f \circ x \neq f \circ y$.

DRAFT: Thursday 2nd February, 2012 16:49

Internal and external hom

If $f : B \rightarrow C$, then f induces a set function $\text{Hom}(A, f) : \text{Hom}(A, B) \rightarrow \text{Hom}(A, C)$ through $\text{Hom}(A, f)(g) = f \circ g$. Similarly, it induces a set function $\text{Hom}(f, A) : \text{Hom}(C, A) \rightarrow \text{Hom}(B, A)$ through $\text{Hom}(f, A)(g) = g \circ f$.

Using this, we have an occasionally enlightening

Proposition An arrow $f : B \rightarrow C$ is

1. a monomorphism if and only if $\text{Hom}(A, f)$ is injective for every object A .
2. an epimorphism if and only if $\text{Hom}(f, A)$ is injective for every object A .
3. a split monomorphism if and only if $\text{Hom}(f, A)$ is surjective for every object A .
4. a split epimorphism if and only if $\text{Hom}(A, f)$ is surjective for every object A .
5. an isomorphism if and only if any one of the following equivalent conditions hold:
 - a) it is both a split epi and a mono.
 - b) it is both an epi and a split mono.
 - c) $\text{Hom}(A, f)$ is bijective for every A .
 - d) $\text{Hom}(f, A)$ is bijective for every A .

For any A, B in a category, the homset is a *set* of morphisms between the objects. For many categories, though, homsets may end up being objects of that category as well.

As an example, the set of all linear maps between two fixed vector spaces is itself a vector space.

Alternatively, the function type `a -> b` is an actual Haskell type, and captures the morphisms of the idealized Haskell category.

We shall return to this situation later, when we are better equipped to give a formal scaffolding to the idea of having elements in objects in a category act as morphisms. For now, we shall introduce the notations $[A \rightarrow B]$ or B^A to denote the *internal* hom - where the morphisms between two objects live as an object of the category. This distinguishes B^A from $\text{Hom}(A, B)$.

To gain a better understanding of the choice of notation, it is worth noting that $|\text{Hom}_{\text{Set}}(A, B)| = |B|^{|A|}$.

Homework

Passing mark requires at least 4 of 11.

1. Suppose g, h are two-sided inverses to f . Prove that $g = h$.
2. (requires some familiarity with analysis) There is a category with object \mathbb{R} (or even all smooth manifolds) and with morphisms smooth (infinitely differentiable) functions $f : \mathbb{R} \rightarrow \mathbb{R}$. Prove that being a bijection does not imply being an isomorphism. Hint: What about $x \mapsto x^3$? Wikipedia definition of smoothness: http://en.wikipedia.org/wiki/Smooth_function. Moral of the definition is that all derivatives and derivatives of derivatives, et.c. are everywhere finite and continuous.
3. (try to do this if you don't do 2) In the category of posets, with order-preserving maps as morphisms, show that not all bijective homomorphisms are isomorphisms. See chapter 1 for details on posets and order-preserving maps, as well as wikipedia links.
4. Consider the partially ordered set P as a category. Prove: every arrow is both monic and epic. Is every arrow thus an isomorphism?
5. What are the terminal and initial objects in a poset? Give an example each of a poset that has both, either and none. Give an example of a poset that has a zero object.
6. What are the terminal and initial objects in the category with objects graphs and morphisms graph homomorphisms? Definition of a graph and a graph homomorphism occurred in chapter 1.
7. Prove that if a category has one zero object, then all initial and all terminal objects are all isomorphic and they are all zero objects.
8. Prove that the composition of two monomorphisms is a monomorphism and that the composition of two epimorphisms is an epimorphism. If $g \circ f$ is monic, do any of g, f have to be monic? If the composition is epic, do any of the factors have to be epic?
9. Verify that the equivalence relation used in defining subobjects really is an equivalence relation. Further verify that this fixes the motivating problem.
10. Describe a representative subcategory each of:

DRAFT: Thursday 2nd February, 2012 16:49

-
- The category of vectorspaces over the reals.
 - The category formed by the preordered set of the integers \mathbb{Z} and the order relation $a \leq b$ if $a|b$. Recall that a preordered set is a set P equipped with a relation \leq that fulfills transitivity and reflexivity, but not necessarily anti-symmetry.
11. * An arrow $f : A \rightarrow A$ in a category C is an *idempotent* if $f \circ f = f$. We say that f is a *split idempotent* if there is some $g : A \rightarrow B, h : B \rightarrow A$ such that $h \circ g = f$ and $g \circ h = 1_B$. Show that in Set , f is idempotent if and only if its image equals its set of fixed points. Show that every idempotent in Set is split. Give an example of a category with a non-split idempotent.

Lecture 3

Functors

These notes cover material dispersed in several places of Awodey. The definition of a functor is on page 8. More on functors and natural transformations comes in sections 7.1-7.2, 7.4-7.5, 7.7-7.10.

Functors

We've spent quite a bit of time talking about categories, and special entities in them - morphisms and objects, and special kinds of them, and properties we can find.

And one of the main messages visible so far is that as soon as we have an algebraic structure, and homomorphisms, this forms a category. More importantly, many algebraic structures, and algebraic theories, can be captured by studying the structure of the category they form.

So obviously, in order to understand Category Theory, one key will be to understand homomorphisms between categories.

Homomorphisms of categories

A category is a graph, so a homomorphism of a category should be a homomorphism of a graph that respect the extra structure. Thus, we are led to the definition:

Definition A *functor* $F : C \rightarrow D$ from a category C to a category D is a graph homomorphism F_0, F_1 between the underlying graphs such that for every object $X \in C_0$:

- $F_1(1_X) = 1_{F_0(X)}$
- $F_1(gf) = F_1(g)F_1(f)$

Note: We shall frequently use F in place of F_0 and F_1 . The context should suffice to tell you whether you are mapping an object or a morphism at any given moment.

On Wikipedia: <http://en.wikipedia.org/wiki/Functor>

Examples

A homomorphism $f : M \rightarrow N$ of monoids is a functor of the corresponding one-object categories $F : C(M) \rightarrow C(N)$. The functor takes the single object to the single object, and acts on morphisms by $F(g) = f(g)$.

A homomorphism $f : P \rightarrow Q$ of posets is a functor $F : C(P) \rightarrow C(Q)$ of the corresponding category. We have $f(x) \leq f(y)$ if $x \leq y$, so if $g \in Hom_P(x, y)$ then $F(g) \in Hom_Q(f(x), f(y))$.

If we pick some basis for every vector space, then this gives us a functor F from $Vect$ to the category with objects integers and morphisms matrices by:

- $F_0(V) = \dim V$
- $F_1(f)$ is the matrix representing f in the matrices chosen.

This example relies on the axiom of choice.

Interpreting functors in Haskell

One example of particular interest to us is the category `Hask`. A functor in `Hask` is something that takes a type, and returns a new type. Not only that, we also require that it takes arrows and return new arrows. So let's pick all this apart for a minute or two.

Taking a type and returning a type means that you are really building a polymorphic type class: you have a family of types parametrized by some type variable. For each type `a`, the functor data `F a = ...` will produce a new type, `F a`. This, really, is all we need to reflect the action of F_0 .

The action of F_1 in turn is recovered by requiring the parametrized type `F a` to implement the `Functor typeclass`. This typeclass requires you to implement a function `fmap :: (a -> b) -> F a -> F b`. This function, as the signature indicates, takes a function `f :: a -> b` and returns a new function `fmap f :: F a -> F b`.

The rules we expect a `Functor` to obey seem obvious: translating from the categorical intuition we arrive at the rules `* fmap id = id` and `* fmap (g . f) = fmap g . fmap f`

Now, the real power of a `Functor` still isn't obvious with this viewpoint. The real power comes in approaching it less categorically.

A Haskell functor is a polymorphic type. In a way, it is an prototypical polymorphic type. We have some type, and we change it, in a meaningful way. And the existence of the `Functor` type-class demands of us that we find a way to translate function applications into the `Functor` image. We can certainly define a boring `Functor`, such as

DRAFT: Thursday 2nd February, 2012 16:49

```
data Boring a = Boring
instance Functor Boring where
    fmap f = const Boring
```

but this is not particularly useful. Almost all `Functor` instances will take your type and include it into something different, something useful. And it does this in a way that allows you to lift functions acting on the type it contains, so that they transform them in their container.

And the choice of words here is deliberate. Functors can be thought of as data containers, their parameters declaring what they contain, and the `fmap` implementation allowing access to the contents. Lists, trees with node values, trees with leaf values, Maybe, Either all are Functors in obvious manners.

```
data List a = Nil | Cons a (List a)
instance Functor List where
    fmap f Nil = Nil
    fmap f (Cons x lst) = Cons (f x) (fmap f lst)
```

```
data Maybe a = Nothing | Just a
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

```
data Either b a = Left b | Right a
instance Functor (Either b) where
    fmap f (Left x) = Left x
    fmap f (Right y) = Right (f y)
```

```
data LeafTree a = Leaf a | Node [LeafTree a]
instance Functor LeafTree where
    fmap f (Node subtrees) = Node (map (fmap f) subtrees)
    fmap f (Leaf x) = Leaf (f x)
```

```
data NodeTree a = Leaf | Node a [NodeTree a]
instance Functor NodeTree where
    fmap f Leaf = Leaf
    fmap f (Node x subtrees) = Node (f x) (map (fmap f) subtrees)
```

The category of categories

We define a category *Cat* by setting objects to be all small categories, and arrows to be all functors between them. Being graph homomorphisms, functors compose, their composition fulfills all requirements on forming a category. It is sometimes useful

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

to argue about a category CAT of all small and most large categories. The issue here is that allowing $CAT \in CAT_0$ opens up for set-theoretical paradoxes.

Isomorphisms in Cat and equivalences of categories

The definition of an isomorphism holds as is in Cat . However, isomorphisms of categories are too restrictive a concept.

To see this, recall the category $Monoid$, where each object is a monoid, and each arrow is a monoid homomorphism. We can form a one-object category out of each monoid, and the method to do this is functorial - i.e. does the right thing to arrows to make the whole process a functor.

Specifically, if $h : M \rightarrow N$ is a monoid homomorphism, we create a new functor $C(h) : C(M) \rightarrow C(N)$ by setting $C(h)_0(*) = *$ and $C(h)_1(m) = h(m)$. This creates a functor from $Monoid$ to Cat . The domain can be further restricted to a full subcategory OOC of Cat , consisting of all the 1-object categories. We can also define a functor $U : OOC \rightarrow Monoid$ by $U(C) = C_1$ with the monoidal structure on $U(C)$ given by the composition in C . For an arrow $F : A \rightarrow B$ we define $U(F) = F_1$.

These functors take a monoid, builds a one-object category, and hits all of them; and takes a one-object category and builds a monoid. Both functors respect the monoidal structures - yet these are not an isomorphism pair. The clou here is that our construction of $C(M)$ from M requires us to choose something for the one object of the category. And choosing different objects gives us different categories.

Thus, the composition CU is not the identity; there is no guarantee that we will pick the object we started with in the construction in C . Nevertheless, we would be inclined to regard the categories $Monoid$ and OOC as essentially the same. The solution is to introduce a different kind of sameness: Definition A functor $F : C \rightarrow D$ is an equivalence of categories if there is a functor $G : D \rightarrow C$ and:

- A family $u_C : C \rightarrow G(F(C))$ of isomorphisms in C indexed by the objects of C , such that for every arrow $f : C \rightarrow C' : G(F(f)) = u_{C'} \circ f \circ u_C^{-1}$.
- A family $u_D : D \rightarrow F(G(D))$ of isomorphisms in D indexed by the objects of D , such that for every arrow $f : D \rightarrow D' : F(G(f)) = u_{D'} \circ f \circ u_D^{-1}$.

The functor G in the definition is called a pseudo-inverse of F .

DRAFT: Thursday 2nd February, 2012 16:49

Natural transformations

The families of morphisms required in the definition of an equivalence show up in more places. Suppose we have two functors $F : A \rightarrow B$ and $G : A \rightarrow B$. Definition A natural transformation $\alpha : F \rightarrow G$ is a family of arrows $\alpha_a : F(a) \rightarrow G(a)$ indexed by the objects of A such that for any arrow $s : a \rightarrow b$ in $G(s) \circ \alpha_a = \alpha_b \circ F(s)$ (draw diagram)

The commutativity of the corresponding diagram is called the *naturality condition* on α , and the arrow α_a is called the *component* of the natural transformation α at the object a .

Given two natural transformations $\alpha : F \rightarrow G$ and $\beta : G \rightarrow H$, we can define a composition $\beta \circ \alpha$ componentwise as $(\beta \circ \alpha)(a) = \beta_a \circ \alpha_a$.

Proposition The composite of two natural transformations is also a natural transformation.

Proposition Given two categories C, D the collection of all functors $C \rightarrow D$ form a category $Func(C, D)$ with objects functors and morphisms natural transformations between these functors.

Note that this allows us to a large degree to use functors to define entities we may otherwise have defined using large and involved definitions. Doing this using the categorical language instead mainly gives us a large number of facts for free: we don't have to verify, say, associativity of composition of functors if we already know them to be functors.

Example Recall our original definition of a graph as two collections and two maps between them. We can define a category `GraphS`: `[[Image:ArightrightarrowsB.gif|A two right arrows B]]` with the two arrows named s and t . It is a finite category with 2 objects, and 4 arrows. Now, a *small graph* can be defined to be just a Functor $GraphS \rightarrow Set$.

In order to define more intricate structures this way, say Categories, or algebraic structures, we'd need more tools - which we shall find in later lectures. This approach to algebraic definition develops into an area called Sketch theory.

The idea, there, is that theories are modelled by specific categories - such as `GraphS` above, and actual instances of the objects they model appear as functors.

With this definition, since a graph is just a functor $GraphS \rightarrow Set$, and we get graph homomorphisms *for free*: a graph homomorphism is just a natural transformation.

And anything we can prove about functors and natural transformations thus immediately gives a corresponding result for graphs and graph homomorphisms.

On Wikipedia, see: http://en.wikipedia.org/wiki/Natural_transformation. Sketch theory, alas, has a painfully incomplete Wikipedia article.

Properties of functors

The process of forming homsets within a category C gives, for any object A , two different functors $\text{Hom}(A, -) : X \mapsto \text{Hom}(A, X)$ and $\text{Hom}(-, A) : X \mapsto \text{Hom}(X, A)$. Functoriality for $\text{Hom}(A, -)$ is easy: $\text{Hom}(A, f)$ is the map that takes some $g : A \rightarrow X$ and transforms it into $fg : A \rightarrow Y$.

Functoriality for $\text{Hom}(-, A)$ is more involved. We can view this as a functor either from C^{op} , or as a different kind of functor. If we just work with C^{op} , then no additional definitions are needed - but we need an intuition for the dual categories.

Alternatively, we introduce a new concept of a contravariant functor. A contravariant functor $F : C \rightarrow D$ is some map of categories, just like a functor is, but such that $F(1[X]) = 1[F(X)]$, as usual, but such that for a $f : A \rightarrow B$, the functor image is some $F(f) : F(B) \rightarrow F(A)$, and the composition is $F(gf) = F(f)F(g)$. The usual kind of functors are named *covariant*.

A functor $F : C \rightarrow D$ is *faithful* if the induced mapping $\text{Hom}_C(A, B) \rightarrow \text{Hom}_D(FA, FB)$ is injective for all $A, B \in C_0$.

A functor $F : C \rightarrow D$ is *full* if the induced mapping $\text{Hom}_C(A, B) \rightarrow \text{Hom}_D(FA, FB)$ is surjective for all $A, B \in C_0$.

Note that a full subcategory is a subcategory so that the embedding functor is both full and faithful.

See also entries in the list of Types of Functors on the Wikipedia page for Functors <http://en.wikipedia.org/wiki/Functor>.

Preservation and reflection

We say that a functor $F : C \rightarrow D$ *preserves* a property P , if whenever P holds in the category C , it does so for the image of F in D .

Thus, the inclusion functor for the category of finite sets into the category of sets preserves both monomorphisms and epimorphisms. Indeed, these properties, in both categories, correspond to injective and surjective functions, respectively; and a surjective (injective) function of finite sets is still surjective (injective) when considered for sets in general.

As another example, consider the category 2 given by $A \rightarrow B$, with the single non-identity arrow named f . All arrows in this category are both monomorphic and epimorphic. We can define a functor $F : 2 \rightarrow \text{Set}$ through $A \mapsto \{1, 2\}$, $B \mapsto \{3, 4\}$ and f mapping to the set function that takes all elements to the value 3. The resulting constant map $F(f)$ is neither epic nor monic, while the morphism f is both.

However, there are properties that functors **do** preserve:

Proposition Every functor preserves isomorphisms.

DRAFT: Thursday 2nd February, 2012 16:49

Proof Suppose $f : X \rightarrow Y$ is an isomorphism with inverse f^{-1} . Then $F(f)$ has inverse $F(f^{-1})$. Indeed, $F(f)F(f^{-1}) = F(ff^{-1}) = F(1_Y) = 1_{F(Y)}$ and $F(f^{-1})F(f) = F(f^{-1}f) = F(1_X) = 1_{F(X)}$. QED

We say that a functor $F : C \rightarrow D$ *reflects* a property P , if whenever $F(f)$ has that property, so does f .

A functor $F : C \rightarrow D$ is *representative* if every object in D is isomorphic to some $F(X)$ for $X \in C_0$.

Homework

Complete homework is by 4 out of 9 complete solutions. Partial credit will be given.

1. Show that the category of vectorspaces is equivalent to the category with objects integers and arrows matrices.
2. Prove the propositions in the section on natural transformations.
3. Prove that `listToMaybe :: [a] -> Maybe a` is a natural transformation from the list functor to the maybe functor. Is `catMaybes` a natural transformation? Between which functors?
4. Find two more natural transformations defined in the standard Haskell library. Find one polymorphic function that is not a natural transformation.
5. Write a functor instance for `data F a = F (Int -> a)`
6. Write a functor instance for `data F a = F ((Int -> a) -> Int)`
7. Write a natural transformation from `Maybe a` to `Either () a`. Is this a natural isomorphism? If so, what is its inverse? If not, why not?
8. Write a natural transformation from `[a]` to `(Maybe a, Maybe a)`. Is this a natural isomorphism? If so, what is its inverse? If not, why not?
9. * Recall that a category is called *discrete* if it has no arrows other than the identities. Show that a small category A is discrete if and only if every set function $A_0 \rightarrow B_0$, for every small category B , is the object part of a unique functor $A \rightarrow B$. Analogously, we define a small category B to be indiscrete if for every small category A , every set function $A_0 \rightarrow B_0$ is the object part of a unique functor $A \rightarrow B$. Characterise indiscrete categories by the objects and arrows they have.

10. * We could write a pretty printer, or XML library, using the following data type as the core data type:

```
data ML a = Tag a (ML a) |
          Str String   |
          Seq [ML a]   |
          Nil
```

With this, we can use a specific string-generating function to generate the tagged marked up text, such as, for instance:

```
prettyprint (Tag tag ml) = "<" ++ show tag ++ ">" ++ prettyprint ml +
prettyprint (Str s) = s
prettyprint (Seq []) = ""
prettyprint (Seq (m:ms)) = prettyprint m ++ "\n" ++ prettyprint (Seq ms)
prettyprint Nil = ""
```

Write an instance of Functor that allows us to apply changes to the tagging type. Then, using the following tagging types:

```
data HTMLTag = HTML | BODY | P | H1 | CLASS String deriving (Show)
data XMLTag = DOCUMENT | HEADING | TEXT deriving (Show)
```

write a function `htmlize :: ML XMLTag -> ML HTMLTag` and use it to generate a html document out of:

```
Tag DOCUMENT
  Seq [
    Tag HEADING
      String "Nobel prize for chromosome find",
    Tag TEXT
      String "STOCKHOLM (Reuters) - Three Americans won the Nobel prize",
    Tag TEXT
      String "Australian-born Elizabeth Blackburn, British-born Jack",
    Tag TEXT
      String "'The discoveries ... have added a new dimension to our understanding",
    Tag TEXT
      String "The trio's work laid the foundation for studies that have",
    Tag TEXT
      String "Work on the enzyme has become a hot area of drug research",
    Tag TEXT
      String "One example, a so-called therapeutic vaccine that targets",
    Tag TEXT
      String "The Chief Executive of Britain's Medical Research Council",
    Tag TEXT
      String "'Their research on chromosomes helped lay the foundation",
  ]
```

Products, coproducts, dualization

Product

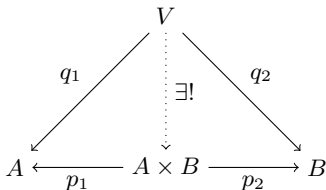
Recall the construction of a cartesian product of two sets: $A \times B = \{(a, b) : a \in A, b \in B\}$. We have functions $p_A : A \times B \rightarrow A$ and $p_B : A \times B \rightarrow B$ extracting the two sets from the product, and we can take any two functions $f : A \rightarrow A'$ and $g : B \rightarrow B'$ and take them together to form a function $f \times g : A \times B \rightarrow A' \times B'$.

Similarly, we can form the type of pairs of Haskell types: `Pair s t = (s, t)`. For the pair type, we have canonical functions `fst :: (s, t) -> s` and `snd :: (s, t) -> t` extracting the components. And given two functions `f :: s -> s'` and `g :: t -> t'`, there is a function `f *** g :: (s, t) -> (s', t')`.

An element of the pair is completely determined by the two elements included in it. Hence, if we have a pair of generalized elements $q_1 : V \rightarrow A$ and $q_2 : V \rightarrow B$, we can find a unique generalized element $q : V \rightarrow A \times B$ such that the projection arrows on this gives us the original elements back.

This argument indicates to us a possible definition that avoids talking about elements in sets in the first place, and we are lead to the

Definition A *product* of two objects A, B in a category C is an object $A \times B$ equipped with arrows $A \xleftarrow{p_1} A \times B \xrightarrow{p_2} B$ such that for any other object V with arrows $A \xleftarrow{q_1} V \xrightarrow{q_2} B$, there is a unique arrow $V \rightarrow A \times B$ such that the diagram



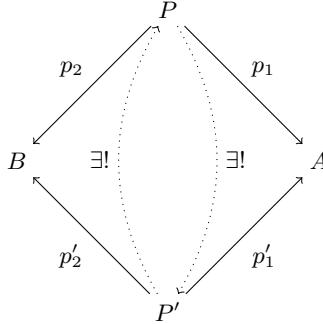
commutes. The diagram $A \xleftarrow{p_1} A \times B \xrightarrow{p_2} B$ is called a *product cone* if it is a diagram of a product with the *projection arrows* from its definition.

In the category of sets, the unique map is given by $q(v) = (q_1(v), q_2(v))$. In the Haskell category, it is given by the combinator $(\&\&\&) :: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c)$.

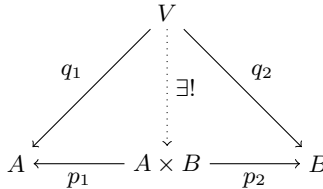
We tend to talk about *the product*. The justification for this lies in the first interesting

Proposition If P and P' are both products for A, B , then they are isomorphic.

Proof Consider the diagram



Both vertical arrows are given by the product property of the two product cones involved. Their compositions are endo-arrows of P, P' , such that in each case, we get a diagram like



with $V = A \times B = P$ (or P'), and $q_1 = p_1, q_2 = p_2$. There is, by the product property, only one endoarrow that can make the diagram work - but both the composition of the two arrows, and the identity arrow itself, make the diagram commute. Therefore, the composition has to be the identity. QED.

We can expand the binary product to higher order products easily - instead of pairs of arrows, we have families of arrows, and all the diagrams carry over to the larger case.

Binary functions

Functions into a product help define the product in the first place, and function as elements of the product. Functions *from* a

product, on the other hand, allow us to put a formalism around the idea of functions of several variables.

So a function of two variables, of types A and B is a function $f :: (A, B) \rightarrow C$. The Haskell idiom for the same thing, $A \rightarrow B \rightarrow C$ as a function taking one argument and returning a function of a single variable; as well as the `curry/uncurry` procedure is tightly connected to this viewpoint, and will reemerge below, as well as when we talk about adjunctions later on.

Coproduct

The product came, in part, out of considering the pair construction. One alternative way to write the `Pair a b` type is:

```
data Pair a b = Pair a b
```

and the resulting type is isomorphic, in Hask, to the product type we discussed above.

This is one of two basic things we can do in a data type declaration, and corresponds to the *record* types in Computer Science jargon.

The other thing we can do is to form a *union* type, by something like

```
data Union a b = Left a | Right b
```

which takes on either a value of type a or of type b , depending on what constructor we use.

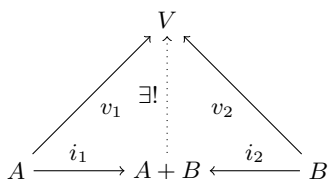
This type guarantees the existence of two functions

```
Left  :: a -> Union a b
Right :: b -> Union a b
```

Similarly, in the category of sets we have the disjoint union $S \coprod T = S \times 0 \cup T \times 1$, which also comes with functions $i_S : S \rightarrow S \coprod T, i_T : T \rightarrow S \coprod T$.

We can use all this to mimic the product definition. The directions of the inclusions indicate that we may well want the dualization of the definition. Thus we define:

Definition A *coproduct* $A + B$ of objects A, B in a category C is an object $A + B$ equipped with arrows $A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$ such that for any other object V with arrows $A \xrightarrow{q_1} V \xleftarrow{q_2} B$, there is a unique arrow $A + B \rightarrow V$ such that the diagram



commutes. The diagram $A \xrightarrow{i_1} A+B \xleftarrow{i_2} B$ is called a *coproduct cocone*, and the arrows are *inclusion arrows*.

For sets, we need to insist that instead of just any $S \times 0$ and $T \times 1$, we need the specific construction taking pairs for the coproduct to work out well. The issue here is that the categorical product is not defined as one single construction, but rather from how it behaves with respect to the arrows involved.

With this caveat, however, the coproduct in `Set` really is the disjoint union sketched above.

For `Hask`, the coproduct is the type construction of `Union` above - more usually written `Either a b`.

And following closely in the dualization of the things we did for products, there is a first

Proposition If C, C' are both coproducts for some pair A, B in a category D , then they are isomorphic.

The proof follows the exact pattern of the corresponding proposition for products.

Algebra of datatypes

Recall from Chapter 3 that we can consider endofunctors as container datatypes. Some of the more obvious such container datatypes include:

```
data 1 a = Empty
data T a = T a
```

These being the data type that has only one single element and the data type that has exactly one value contained.

Using these, we can generate a whole slew of further datatypes. First off, we can generate a data type with any finite number of elements by $n = 1 + 1 + \dots + 1$ (n times). Remember that the coproduct construction for data types allows us to know which summand of the coproduct a given part is in, so the single elements in all the 1s in the definition of `n` here are all distinguishable, thus giving the final type the required number of elements. Of note among these is the data type `Bool = 2` - the Boolean data type, characterized by having exactly two elements.

Furthermore, we can note that $1 \times T = T$, with the isomorphism given by the maps

DRAFT: Thursday 2nd February, 2012 16:49

```
f (Empty, T x) = T x
g (T x) = (Empty, T x)
```

Thus we have the capacity to *add* and *multiply* types with each other. We can verify, for any types A, B, C $A \times (B + C) = A \times B + A \times C$

We can thus make sense of types like $T^3 + 2T^2$ (either a triple of single values, or one out of two tagged pairs of single values).

This allows us to start working out a calculus of data types with versatile expression power. We can produce recursive data type definitions by using equations to define data types, that then allow a direct translation back into Haskell data type definitions, such as:

```
List = 1 + T × List
BinaryTree = T + T × BinaryTree × BinaryTree
TernaryTree = T + T × TernaryTree × TernaryTree × TernaryTree
GenericTree = T + T × (List ◦ GenericTree)
```

The real power of this way of rewriting types comes in the recognition that we can use algebraic methods to reason about our data types. For instance:

```
List = 1 + T * List
      = 1 + T * (1 + T * List)
      = 1 + T * 1 + T * T * List
      = 1 + T + T * T * List
```

so a list is either empty, contains one element, or contains at least two elements. Using, though, ideas from the theory of power series, or from continued fractions, we can start analyzing the data types using steps on the way that seem completely bizarre, but arriving at important property. Again, an easy example for illustration:

```
List = 1 + T * List           -- and thus
List - T * List = 1          -- even though (-) doesn't make sense for data types
(1 - T) * List = 1           -- still ignoring that (-)...
List = 1 / (1 - T)           -- even though (/) doesn't make sense for data types
      = 1 + T + T*T + T*T*T + ... -- by the geometric series identity
```

and hence, we can conclude - using formally algebraic steps in between - that a list by the given definition consists of either an empty list, a single value, a pair of values, three values, etc.

At this point, I'd recommend anyone interested in more perspectives on this approach to data types, and thinks one may do with them, to read the following references:

DRAFT: Thursday 2nd February, 2012 16:49
Please do not circulate

Blog posts and Wikipages

The ideas in this last section originate in a sequence of research papers from Conor McBride - however, these are research papers in logic, and thus come with all the quirks such research papers usually carry. Instead, the ideas have been described in several places by various blog authors from the Haskell community - which make for a more accessible but much less strict read.

- <http://en.wikibooks.org/wiki/Haskell/Zipper> -- On zippers, and differentiating types
- <http://blog.lab49.com/archives/3011> -- On the polynomial data type calculus
- <http://blog.lab49.com/archives/3027> -- On differentiating types and zippers
- <http://comonad.com/reader/2008/generatingfunctorology/> - Different recursive type constructions
- <http://strictlypositive.org/slicing-jpgs/> -- Lecture slides for similar themes.
- <http://blog.sigfpe.com/2009/09/finite-differences-of-types.html> -- Finite differences of types - generalizing the differentiation approach.
- <http://homepage.mac.com/sigfpe/Computing/fold.html> -- Develops the underlying theory for our algebra of datatypes in some detail.

Homework

Complete points for this homework consists of 4 out of 5 exercises. Partial credit is given.

1. What are the products in the category $C(P)$ of a poset P ? What are the coproducts?
2. Prove that any two coproducts are isomorphic.
3. Prove that any two exponentials are isomorphic.
4. Write down the type declaration for at least two of the example data types from the section of the algebra of datatypes, and write a `Functor` implementation for each.
5. * Read up on Zippers and on differentiating data structures. Find the derivative of `List`, as defined above. Prove that $\partial List = List \times List$. Find the derivatives of `BinaryTree`, and of `GenericTree`.

Limits and colimits

Cartesian Closed Categories and typed lambda-calculus

A category is said to *have pairwise products* if for any objects A, B , there is a product object $A \times B$.

A category is said to *have pairwise coproducts* if for any objects A, B , there is a coproduct object $A + B$.

Recall when we talked about internal homs in Lecture 2. We can now define what we mean, formally, by the concept:

Definition An object C in a category D is an *internal hom object* or an *exponential object* $[A \rightarrow B]$ or B^A if it comes equipped with an arrow $ev : [A \rightarrow B] \times A \rightarrow B$, called the *evaluation arrow*, such that for any other arrow $f : C \times A \rightarrow B$, there is a unique arrow $\lambda f : C \rightarrow [A \rightarrow B]$ such that the composite

$$C \times A \xrightarrow{\lambda f \times 1_A} [A \rightarrow B] \times A \xrightarrow{ev} B$$

is f .

The idea here is that with something in an exponential object, and something in the source of the arrows we imagine live inside the exponential, we can produce the evaluation of the arrow at the source to produce something in the target. Using global elements, this reasoning comes through in a more natural manner: given $f : 1 \rightarrow [A \rightarrow B]$ and $x : 1 \rightarrow A$ we can produce the global element $f(x) = ev \circ f \times x : 1 \rightarrow B$. Furthermore, we can always produce something in the exponential whenever we have something that looks as if it should be there.

And with this we can define

Definition A category C is a *Cartesian Closed Category* or a CCC if:

1. C has a terminal object 1
2. Each pair of objects $A, B \in C_0$ has a product $A \times B$ and projections $p_1 : A \times B \rightarrow A$, $p_2 : A \times B \rightarrow B$.
3. For every pair $A, B \in C_0$ of objects, there is an exponential object $[A \rightarrow B]$ with an evaluation map $[A \rightarrow B] \times A \rightarrow B$.

Currying

Note that the exponential as described here is exactly what we need in order to discuss the Haskell concept of multi-parameter functions. If we consider the type of a binary function in Haskell:

```
binFunction :: a -> a -> a
```

This function really lives in the Haskell type $a \rightarrow (a \rightarrow a)$, and thus is an element in the repeated exponential object $[A \rightarrow [A \rightarrow A]]$. Evaluating once gives us a single-parameter function, the first parameter consumed by the first evaluation, and we can evaluate a second time, feeding in the second parameter to get an end result from the function.

On the other hand, we can feed in both values at once, and get

```
binFunction' :: (a,a) -> a
```

which lives in the exponential object $[A \times A \rightarrow A]$.

These are genuinely different objects, but they seem to do the same thing: consume two distinct values to produce a third value. The resolution of the difference lies, again, in a recognition from Set theory: there is an isomorphism

$$\text{Hom}(S, \text{Hom}(T, V)) = \text{Hom}(S \times T, V)$$

which we can use as inspiration for an isomorphism $\text{Hom}(S, [T \rightarrow V]) = \text{Hom}(S \times T, V)$ valid in Cartesian Closed Categories.

Typed lambda-calculus

The lambda-calculus, and later the *typed* lambda-calculus both act as foundational bases for computer science, and computer programming in particular. The idea in both is that everything is a function, and we can reduce the act of programming to function application; which in turn can be analyzed using expression rewriting rules that encapsulate the act of computation in a sequence of formal rewrites.

Definition A *typed lambda-calculus* is a formal theory with *types*, *terms*, *variables* and *equations*. Each term a has a type A associated to it, and we write $a : A$ or $a \in A$. The system is subject to a sequence of rules:

1. There is a type 1. *Hence, the empty lambda calculus is excluded.*
2. If A, B are types, then so are $A \times B$ and $[A \rightarrow B]$. *These are, initially, just additional symbols, not imbued with the associations we usually give the symbols used.*

DRAFT: Thursday 2nd February, 2012 16:49

-
3. There is a term $*$: 1. Hence, the lambda calculus without any terms is excluded.
 4. For each type A , there is an infinite (countable) supply of terms $x_A^i : A$.
 5. If $a : A, b : B$ are terms, then there is a term $(a, b) : A \times B$.
 6. If $c : A \times B$ then there are terms $proj_1(c) : A, proj_2(c) : B$.
 7. If $a : A$ And $f : [A \rightarrow B]$, then there is a term $fa : B$.
 8. If $x : A$ is a variable and $\phi(x) : B$ is a term, then there is a $\lambda_{x \in A} \phi(x) : [A \rightarrow B]$. Note that here, $\phi(x)$ is a meta-expression, meaning we have SOME lambda-calculus expression that may include the variable x .
 9. There is a relation $a =_X a'$ for each set of variables X that occur freely in either a or a' . This relation is reflexive, symmetric and transitive. Recall that a variable is free in a term if it is not in the scope of a λ -expression naming that variable.
 10. If $a : 1$ then $a =_{\{\}} *$. In other words, up to lambda-calculus equality, there is only one value of type $*$.
 11. If $X \subseteq Y$, then $a =_X a'$ implies $a =_Y a'$. Binding more variables gives less freedom, not more, and thus cannot suddenly make equal expressions differ.
 12. $a =_X a'$ implies $fa =_X fa'$.
 13. $f =_X f'$ implies $fa =_X fa'$. So equality plays nice with function application.
 14. $\phi(x) =_{X \cup \{x\}} \phi'(x)$ implies $\lambda_x \phi(x) =_X \lambda_x \phi'(x)$. Equality behaves well with respect to binding variables.
 15. $proj_1(a, b) =_X a, proj_2(a, b) =_X b, c =_X (proj_1(c), proj_2(c))$ for all a, b, c, X .
 16. $\lambda_x \phi(x)a =_X \phi(a)$ if a is substitutable for x in $\phi(x)$ and $\phi(a)$ is what we get by substituting each occurrence of x by a in $\phi(x)$. A term is **substitutable** for another if by performing the substitution, no occurrence of any variable in the term becomes bound,
 17. $\lambda_{x \in A} fx =_X f$, provided $x \notin X$.
 18. $\lambda_{x \in A} \phi(x) =_X \lambda_{x' \in A} \phi(x')$ if x' is substitutable for x in $\phi(x)$ and each variable is not free in the other expression.

Note that $=_x$ is *just a symbol*. The axioms above give it properties that work a lot like equality, but two lambda calculus-equal terms are not equal unless they are identical. However, $a =_x b$ tells us that in any *model* of this lambda calculus - where terms, types, et.c. are replaced with actual things (mathematical objects, say, or a programming language semantics embedding typed lambda calculus) - *then* the things given by translating a and b into the model should end up being equal.

Any actual realization of typed lambda calculus is bound to have more rules and equalities than the ones listed here.

With these axioms in front of us, however, we can see how lambda calculus and Cartesian Closed Categories fit together: We can go back and forth between the two concepts in a natural manner:

Lambda to CCC Given a typed lambda calculus L , we can define a CCC $C(L)$. Its objects are the types of L . An arrow from A to B is an equivalence class (under $=_{\{x\}}$) of terms of type B , free in a single variable $x : A$.

We need the equivalence classes because for any variable $x : A$, we want $\lambda_x x : 1 \rightarrow [A \rightarrow A]$ to be the global element of $[A \rightarrow A]$ corresponding to the identity arrow. Hence, that variable must itself correspond to an identity arrow.

And then the rules for the various constructions enumerated in the axioms correspond closely to what we need to prove the resulting category to be cartesian closed.

CCC to Lambda To go in the other direction, starting out with a Cartesian Closed Category and finding a typed lambda calculus corresponding to it, we construct its *internal language*.

Given a CCC C , we can assume that we have chosen, somehow, one actual product for each finite set of factors. Thus, both all products and all projections are well defined entities, with no remaining choice to determine them.

The *types* of the internal language $L(C)$ are just the objects of C . The existence of products, exponentials and terminal object covers axioms 1-2. We can assume the existence of variables for each type, and the remaining axioms correspond to definition and behaviour of the terms available.

Using the properties of a CCC, it is at this point possible to prove a resulting equivalence of categories $C(L(C)) = C$, and similarly, with suitable definitions for what it means for formal languages to be equivalent, one can also prove for a typed lambda-calculus L that $L(C(L)) = L$.

--- More on this subject can be found in:

DRAFT: Thursday 2nd February, 2012 16:49

-
- Lambek & Scott: *Aspects of higher order categorical logic and Introduction to higher order categorical logic*

More importantly, by stating λ -calculus in terms of a CCC instead of in terms of *terms* and *rewriting rules* is that you can escape worrying about variable clashes, alpha reductions and composability - the categorical translation ignores, at least superficially, the variables, reduces terms with morphisms that have equality built in, and provides associative composition *for free*.

At this point, I'd recommend reading more on Wikipedia http://en.wikipedia.org/wiki/Lambda_calculus and http://en.wikipedia.org/wiki/Cartesian_closed_category, as well as in Lambek & Scott: *Introduction to Higher Order Categorical Logic*. The book by Lambek & Scott goes into great depth on these issues, but may be less than friendly to a novice.

Limits and colimits

One *design pattern*, as it were, that we have seen occur over and over in the definitions we've seen so far is for there to be some object, such that for every other object around, certain morphisms have unique existence.

We saw it in terminal and initial objects, where there's a unique map from or to **every** other object. And in products/coproducts where a wellbehaved map, capturing any pair of maps has unique existence. And finally, above, in the CCC characterization of the internal hom, we had a similar uniqueness requirement for the lambda map.

One thing we can notice is that the isomorphisms theorems for all these cases look very similar to each other: in each isomorphism proof, we produce the uniquely existing morphisms, and prove that their uniqueness and their other properties force the maps to really be isomorphisms.

Now, category theory has a philosophy slightly similar to design patterns - if we see something happening over and over, we'll want to generalize it. And there are generalizations available for these!

Diagrams, cones and limits

Definition A *diagram* D of the shape of an index category J (often finite or countable), in a category C is just a functor $D : J \rightarrow C$. Objects in J will be denoted by i, j, k, \dots and their images in C by D_i, D_j, D_k, \dots

This underlines that when we talk about diagrams, we tend to think of them less as just functors, and more as their images -

the important part of a diagram D is the objects and their layout in C , and not the process of going to C from D .

Definition A *cone* over a diagram D in a category C is some object C equipped with a family $c_i : C \rightarrow D_i$ of arrows, one for each object in J , such that for each arrow $\alpha : i \rightarrow j$ in J , the following diagram

$$\begin{array}{ccc} C & & \\ \downarrow c_i & \searrow c_j & \\ D_i & \xrightarrow{D_\alpha} & D_j \end{array}$$

commutes, or in equations, $D_\alpha c_i = c_j$.

A *morphism* $f : (C, c_i) \rightarrow (C', c'_i)$ of cones is an arrow $f : C \rightarrow C'$ such that each triangle

$$\begin{array}{ccc} C & \xrightarrow{f} & C' \\ \downarrow c_i & \searrow c'_i & \\ D_i & & \end{array}$$

commutes, or in equations, such that $c_j = c'_j f$.

This defines a category of cones, that we shall denote by $\text{Cone}(D)$. And we define, hereby:

Definition The *limit* of a diagram D in a category C is a terminal object in $\text{Cone}(D)$. We often denote a limit by

$$p_i : \lim_{\leftarrow j} D_j \rightarrow D_i$$

so that the map from the limit object $\lim_{\leftarrow j} D_j$ to one of the diagram objects D_i is denoted by p_i .

The limit being terminal in the category of cones nails down once and for all the uniqueness of any map into it, and the isomorphism of any two terminal objects carries over to a proof once and for all for the limit case.

Specifically, since the morphisms of cones are morphisms in C , and composition is carried straight over, so proving a map is an isomorphism in the cone category implies it is one in the target category as well.

Definition A category C has all (finite) limits if all diagrams (of finite shape) have limit objects defined for them.

DRAFT: Thursday 2nd February, 2012 16:49

Limits we've already seen

The terminal object of a category is the limit object of an empty diagram. Indeed, it is an object, with *no* specified maps to *no* other objects, such that every other object that also maps to the same empty set of objects - which is to say all other objects - have a uniquely determined map to the limit object.

The product of some set of objects is the limit object of the diagram containing all these objects and no arrows; a diagram of the shape of a discrete category. The condition here becomes the requirement of maps to all factors so any other cone factors through these maps.

To express the exponential as a limit, we need to go to a different category than the one we started in. Take the category with objects given by morphisms $X \times Y \rightarrow Z$ for fixed objects Y, Z , and morphisms given by morphisms $X \times Y \rightarrow X' \times Y$ commuting with the 'objects' they run between and fixing Y . The exponential is a terminal object in this category.

Adding further arrows to diagrams amounts to adding further conditions on the products, as the maps from the product to the diagram objects need to factor through any arrows present in the diagram.

These added relations, however, is exactly what trips things up in Haskell. The idealized Haskell category does not have even all finite limits. At the core of the issue here is the lack of dependent types: there is no way for the type system to guarantee equations, and hence only the trivial limits - the products - can be guaranteed by the Haskell type checked.

In order to get that kind of guarantees, the type checker would need an implementation of a Dependent type(citation), something that can be simulated in several ways, but is not (yet) an actual part of Haskell. Other languages, however, cover this - most notably Epigram, Agda and Cayenne - which the latter is much stronger influenced by constructive type theory and category theory even than Haskell.

The kind of equations that show up in a limit, however, could be thought of as invariants for the type - and thus something that can be tested for. The resulting equations can be plugged into a testing framework - such as QuickCheck(citation) to verify that the invariants hold under the functions applied.

Colimits

The dual concept to a limit is defined using the dual to the cones:

DRAFT: Thursday 2nd February, 2012 16:49
Please do not circulate

Definition A *cocone* over a diagram $D : J \rightarrow C$ is an object C with arrows $c_j : D_j \rightarrow C$ such that for each arrow $\alpha : i \rightarrow j$ in J , the following diagram

$$\begin{array}{ccc} D_i & \xrightarrow{D_\alpha} & D_j \\ c_i \downarrow & \searrow c_j & \\ C & & \end{array}$$

commutes, or in equations, such that $c_j D_\alpha = c_i$.

A *morphism* $f : (C, c_i) \rightarrow (C', c'_i)$ of cocones is an arrow $f : C \rightarrow C'$ such that each triangle

$$\begin{array}{ccc} D_i & & \\ c_i \downarrow & \searrow c'_i & \\ C & \xrightarrow{f} & C' \end{array}$$

commutes, or in equations, such that $c_j = c'_j f$.

Just as with the category of cones, this yields a category of cocones, that we denote by $\text{Cocone}(D)$, and with this we define:

Definition The *colimit* of a diagram $D : J \rightarrow C$ is an initial object in $\text{Cocone}(D)$.

We denote the colimit by

$$i_i : D_i \rightarrow \lim_{\rightarrow j} D_j$$

so that the map from one of the diagram objects D_i to the colimit object $\lim_{\rightarrow j} D_j$ is denoted by i_i .

Again, the isomorphism results for coproducts and initial objects follow from that for the colimit, and the same proof ends up working for all colimits.

And again, we say that a category *has (finite) colimits* if every (finite) diagram admits a colimit.

Colimits we've already seen

The initial object is the colimit of the empty diagram.

The coproduct is the colimit of the discrete diagram.

For both of these, the argument is almost identical to the one in the limits section above.

Homework

Credit will be given for up to 4 of the 6 exercises.

1. Prove that currying/uncurrying are isomorphisms in a CCC.
Hint: the map $f \mapsto \lambda f$ is a map $Hom(C \times A, B) \rightarrow Hom(C, [A \rightarrow B])$.
2. Prove that in a CCC λev is $\lambda ev = 1_{[A \rightarrow B]} : [A \rightarrow B] \rightarrow [A \rightarrow B]$.
3. What is the limit of a diagram of the shape of the category 2 ?
4. Is the category of Sets a CCC? Prove it.
5. Is the category of vector spaces a CCC? Prove it.
6. * Implement a typed lambda calculus as an EDSL in Haskell.

Pairs of constructions

Useful limits and colimits

With the tools of limits and colimits at hand, we can start using these to introduce more category theoretical constructions - and some of these turn out to correspond to things we've seen in other areas.

Possibly among the most important are the equalizers and coequalizers (with kernel (nullspace) and images as special cases), and the pullbacks and pushouts (with which we can make explicit the idea of inverse images of functions).

One useful theorem to know about is:

Theorem The following are equivalent for a category C :

- C has all finite limits.
- C has all finite products and all equalizers.
- C has all pullbacks and a terminal object.

Also, the following dual statements are equivalent:

- C has all finite colimits.
- C has all finite coproducts and all coequalizers.
- C has all pushouts and an initial object.

For this theorem, we can replace *finite* with any other cardinality in every place it occurs, and we will still get a valid theorem.

Equalizer, coequalizer

Consider the *equalizer diagram*:

$$A \begin{array}{c} \xrightarrow{f} \\ \xRightarrow{g} \end{array} B$$

A limit over this diagram is an object C and arrows to all diagram objects. The commutativity conditions for the arrows defined force for us $fp_A = p_B = gp_A$, and thus, keeping this enforced equation in mind, we can summarize the cone diagram as:

$$C \xrightarrow{p_A} A \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{g} \end{array} B$$

Now, the limit condition tells us that this is the least restrictive way we can map into A with some map p such that $fp = gp$, in that every other way we could map in that way will factor through this way.

As usual, it is helpful to consider the situation in \mathbf{Set} to make sense of any categorical definition: and the situation there is helped by the generalized element viewpoint: the limit object C is one representative of a subobject of A that for the case of \mathbf{Set} contains all $x \in A : f(x) = g(x)$.

Hence the word we use for this construction: the limit of the diagram above is the *equalizer of f, g* . It captures the idea of a maximal subset unable to distinguish two given functions, and it introduces a categorical way to define things by equations we require them to respect.

One important special case of the equalizer is the *kernel*: in a category with a null object, we have a distinguished, unique, member 0 of any homset given by the compositions of the unique arrows to and from the null object. We define *the kernel $\text{Ker}(f)$* of an arrow f to be the equalizer of $f, 0$. Keeping in mind the arrow-centric view on categories, we tend to denote the arrow from $\text{Ker}(f)$ to the source of f by $\text{ker}(f)$.

In the category of vector spaces, and linear maps, the map 0 really is the constant map taking the value 0 everywhere. And the kernel of a linear map $f : U \rightarrow V$ is the equalizer of $f, 0$. Thus it is some vector space W with a map $i : W \rightarrow U$ such that $fi = 0i = 0$, and any other map that fulfills this condition factors through W . Certainly the vector space $\{u \in U : f(u) = 0\}$ fulfills the requisite condition, nothing larger will do, since then the map composition wouldn't be 0 , and nothing smaller will do, since then the maps factoring this space through the smaller candidate would not be unique.

Hence, $\text{Ker}(f) = \{u \in U : f(u) = 0\}$ just like we might expect.

Dually, we get the *coequalizer* as the colimit of the equalizer diagram.

A coequalizer

$$A \begin{array}{c} \xrightarrow{f} \\ \xrightarrow{g} \end{array} B \xrightarrow{i_B} Q$$

has to fulfill that $i_B f = i_A = i_B g$. Thus, writing $q = i_B$, we get an object with an arrow (actually, an epimorphism out of B) that identifies f and g . Hence, we can think of $i_B : B \rightarrow Q$ as catching the notion of inducing equivalence classes from the functions.

This becomes clear if we pick out one specific example: let $R \subseteq X \times X$ be an equivalence relation, and consider the diagram

$$R \begin{array}{c} \xrightarrow{r_1} \\ \xrightarrow{r_2} \end{array} X$$

where r_1 and r_2 are given by the projection of the inclusion of the relation into the product onto either factor. Then, the coequalizer of this setup is an object X/R such that whenever $x \sim_R y$, then $q(x) = q(y)$.

Pullbacks

The preimage $f^{-1}(T)$ of a subset $T \subseteq S$ along a function $f : U \rightarrow S$ is a maximal subset $V \subseteq U$ such that for every $v \in V : f(v) \in T$.

We recall that subsets are given by (equivalence classes of) monics, and thus we end up being able to frame this in purely categorical terms. Given a diagram like this:

$$\begin{array}{ccc} V & \overset{\bar{f}}{\dashrightarrow} & T \\ j \downarrow & & \downarrow i \\ U & \xrightarrow{f} & S \end{array}$$

where i is a monomorphism representing the subobject, we need to find an object V with a monomorphism injecting it into U such that the map $\bar{f}j : V \rightarrow S$ factors through T . Thus we're looking for dotted maps making the diagram commute, in a universal manner.

The maximality of the subobject means that any other subobject of U that can be factored through T should factor through V .

Suppose U, V are subsets of some set W . Their intersection $U \cap V$ is a subset of U , a subset of V and a subset of W , maximal with this property.

Translating into categorical language, we can pick representatives for all subobjects in the definition, we get a diagram with all monomorphisms:

$$\begin{array}{ccc}
 U \cap V & \xrightarrow{\bar{j}} & V \\
 \downarrow \bar{i} & & \downarrow i \\
 U & \xrightarrow{j} & W
 \end{array}$$

where we need the inclusion of $U \cap V$ into W over U to be the same as the inclusion over V .

Definition A *pullback* of two maps $A \xrightarrow{f} C \xleftarrow{g} B$ is the limit of these two maps, thus:

$$\begin{array}{ccc}
 P & \xrightarrow{\bar{g}} & A \\
 \downarrow \bar{f} & \searrow & \downarrow f \\
 B & \xrightarrow{g} & C
 \end{array}$$

By the definition of a limit, this means that the pullback is an object P with maps $\bar{f} : P \rightarrow B$, $\bar{g} : P \rightarrow A$ and $f\bar{g} = g\bar{f} : P \rightarrow C$, such that any other such object factors through this.

For the diagram $U \xrightarrow{f} S \xleftarrow{i} T$, with $i : T \rightarrow S$ one representative monomorphism for the subobject, we get precisely the definition above for the inverse image.

For the diagram $U \rightarrow W \leftarrow V$ with both map monomorphisms representing their subobjects, the pullback is the intersection.

Pushouts

Often, especially in geometry and algebra, we construct new structures by gluing together old structures along substructures. Possibly the most popularly known example is the Möbius band: we take a strip of paper, twist it once and glue the ends together.

Similarly, in algebraic contexts, we can form *amalgamated products* that do roughly the same.

All these are instances of the dual to the pullback:

Definition A *pushout* of two maps $A \xleftarrow{f} C \xrightarrow{g} B$ is the co-limit of these two maps, thus:

$$\begin{array}{ccc}
 C & \xrightarrow{g} & A \\
 \downarrow f & \searrow & \downarrow \bar{f} \\
 B & \xrightarrow{\bar{g}} & D
 \end{array}$$

Hence, the pushout is an object D such that C maps to the same place both ways, and so that, contingent on this, it behaves much like a coproduct.

Free and forgetful functors

Recall how we defined a free monoid as all strings of some alphabet, with concatenation of strings the monoidal operation. And recall how we defined the free category on a graph as the category of paths in the graph, with path concatenation as the operation.

The reason we chose the word *free* to denote both these cases is far from a coincidence: by this point nobody will be surprised to hear that we can unify the idea of generating the most general object of a particular algebraic structure into a single categorical idea.

The idea of the free constructions, classically, is to introduce as few additional relations as possible, while still generating a valid object of the appropriate type, given a set of generators we view as placeholders, as symbols. Having a minimal amount of relations allows us to introduce further relations later, by imposing new equalities by mapping with surjections to other structures.

One of the first observations in each of the cases we can do is that such a map ends up being completely determined by where the generators go - the symbols we use to generate. And since the free structure is made to fulfill the axioms of whatever structure we're working with, these generators combine, even after mapping to some other structure, in a way compatible with all structure.

To make solid categorical sense of this, however, we need to couple the construction of a free algebraic structure from a set (or a graph, or...) with another construction: we can define the *forgetful functor* from monoids to sets by just picking out the elements of the monoid as a set; and from categories to graph by just picking the underlying graph, and forgetting about the compositions of arrows.

Now we have what we need to pinpoint just what kind of a functor the *free widget generated by*-construction does. It's a functor $F : C \rightarrow D$, coupled with a forgetful functor $U : D \rightarrow C$ such that any map $S \rightarrow U(N)$ in C induces one unique mapping $F(S) \rightarrow N$ in D .

For the case of monoids and sets, this means that if we take our generating set, and map it into the set of elements of another monoid, this generates a unique mapping of the corresponding monoids.

This is all captured by a similar kind of diagrams and uniquely existing maps argument as the previous object or morphism properties were defined with. We'll show the definition for the example of monoids.

Definition A *free monoid* on a generating set X is a monoid $F(X)$ such that there is an inclusion $i_X : X \rightarrow UF(X)$ and for every function $f : X \rightarrow U(M)$ for some other monoid M , there is a unique homomorphism $g : F(X) \rightarrow M$ such that $f = U(g)i_X$, or in other words such that this diagram commutes:

$$\begin{array}{ccc} X & & \\ \downarrow i_X & \searrow f & \\ UF(X) & \xrightarrow{\quad \exists! U(g) \quad} & U(M) \end{array}$$

We can construct a map $\phi : \text{Hom}_{\text{Mon}}(F(X), M) \rightarrow \text{Hom}_{\text{Set}}(X, U(M))$ by $\phi : g \mapsto U(g) \circ i_X$. The above definition says that this map is an isomorphism.

Adjunctions

Modeling on the way we construct free and forgetful functors, we can form a powerful categorical concept, which ends up generalizing much of what we've already seen - and also leads us on towards monads.

We draw on the definition above of free monoids to give a preliminary definition. This will be replaced later by an equivalent definition that gives more insight.

Definition A pair of functors,

$$C \begin{array}{c} \xrightarrow{F} \\ \xleftarrow{U} \end{array} D$$

is called an *adjoint pair* or an *adjunction*, with F called the *left adjoint* and U called the *right adjoint* if there is natural transformation $\eta : 1 \rightarrow UF$, and for every $f : A \rightarrow U(B)$, there is a unique $g : F(A) \rightarrow B$ such that the diagram below commutes.

$$\begin{array}{ccc} A & & \\ \downarrow \eta_A & \searrow f & \\ UF(A) & \xrightarrow{\quad \exists! U(g) \quad} & U(B) \end{array}$$

The natural transformation η is called the *unit* of the adjunction.

This definition, however, has a significant amount of asymmetry: we can start with some $f : A \rightarrow U(B)$ and generate a $g : F(A) \rightarrow B$, while there are no immediate guarantees for the other direction. However, there is a proposition we can prove leading us to a more symmetric statement:

Proposition For categories and functors

$$C \begin{matrix} \xrightarrow{F} \\ \xleftarrow{U} \end{matrix} D$$

the following conditions are equivalent:

1. F is left adjoint to U .
2. For any $c \in C_0, d \in D_0$, there is an isomorphism $\phi : \text{Hom}_D(Fc, d) \rightarrow \text{Hom}_C(c, Ud)$, natural in both c and d .

moreover, the two conditions are related by the formulas

- $\phi(g) = U(g) \circ \eta_c$
- $\eta_c = \phi(1_{Fc})$

Proof sketch For (1 implies 2), the isomorphism is given by the end of the statement, and it is an isomorphism exactly because of the unit property - viz. that every $f : A \rightarrow U(B)$ generates a unique $g : F(A) \rightarrow B$.

Naturality follows by building the naturality diagrams

$$\begin{array}{ccc}
 \text{Hom}_D(Fc, d) & \xrightarrow{\phi} & \text{Hom}_C(c, Ud) \\
 \text{Hom}(Ff, d) \downarrow & & \downarrow \text{Hom}(f, Ud) \\
 \text{Hom}_D(Fc', d) & \xrightarrow{\phi} & \text{Hom}_C(c', Ud) \\
 \\
 \text{Hom}_D(Fc, d) & \xrightarrow{\phi} & \text{Hom}_C(c, Ud) \\
 \text{Hom}(Fc, g) \downarrow & & \downarrow \text{Hom}(c, Ug) \\
 \text{Hom}_D(Fc, d') & \xrightarrow{\phi} & \text{Hom}_C(c, Ud')
 \end{array}$$

and chasing through with a $f : Fc \rightarrow d$.

For (2 implies 1), we start out with a natural isomorphism ϕ . We find the necessary natural transformation η_c by considering $\phi : \text{Hom}(Fc, Fc) \rightarrow \text{Hom}(c, UFc)$.

QED.

By dualizing the proof, we get the following statement:

Proposition For categories and functors

$$C \begin{array}{c} \xrightarrow{F} \\ \xleftarrow{U} \end{array} D$$

the following conditions are equivalent:

1. For any $c \in C_0, d \in D_0$, there is an isomorphism $\phi : \text{Hom}_D(Fc, d) \rightarrow \text{Hom}_C(c, Ud)$, natural in both c and d .
2. There is a natural transformation $\epsilon : FU \rightarrow 1_D$ with the property that for any $g : F(c) \rightarrow d$ there is a unique $f : c \rightarrow U(d)$ such that $g = \epsilon_D \circ F(f)$, as in the diagram

$$\begin{array}{ccc} Fc & & \\ \exists! Ff \downarrow \text{dotted} & \searrow g & \\ FU(d) & \xrightarrow{\epsilon_d} & d \end{array}$$

moreover, the two conditions are related by the formulas

- $\psi(f) = \epsilon_D \circ F(f)$
- $\epsilon_d = \psi(1_{Ud})$

where $\psi = \phi^{-1}$.

Hence, we have an equivalent definition with higher generality, more symmetry and more *horsepower*, as it were:

Definition An *adjunction* consists of functors

$$C \begin{array}{c} \xrightarrow{F} \\ \xleftarrow{U} \end{array} D$$

and a natural isomorphism

$$\text{Hom}_D(Fc, d) \begin{array}{c} \xleftarrow{\phi} \\ \xrightarrow{\psi} \end{array} \text{Hom}_C(c, Ud)$$

The *unit* η and the *counit* ϵ of the adjunction are natural transformations given by:

- $\eta : 1_C \rightarrow UF : \eta_c = \phi(1_{Fc})$
- $\epsilon : FU \rightarrow 1_D : \epsilon_d = \psi(1_{Ud})$.

DRAFT: Thursday 2nd February, 2012 16:49

Some of the examples we have had difficulties fitting into the limits framework show up as adjunctions:

The *free* and *forgetful* functors are adjoints; and indeed, a more natural definition of what it means to be free is that it is a left adjoint to some forgetful functor.

Curry and uncurry, in the definition of an exponential object are an adjoint pair. The functor $- \times A : X \mapsto X \times A$ has right adjoint $-^A : Y \mapsto Y^A$.

Notational aid

One way to write the adjoint is as a *bidirectional rewrite rule*:

$$\frac{F(X) \rightarrow Y}{X \rightarrow G(Y)},$$

where the statement is that the hom sets indicated by the upper and lower arrow, respectively, are transformed into each other by the unit and counit respectively. The left adjoint is the one that has the functor application on the left hand side of this diagram, and the right adjoint is the one with the functor application to the right.

Homework

Complete homework is 6 out of 11 exercises.

1. Prove that an equalizer is a monomorphism.
2. Prove that a coequalizer is an epimorphism.
3. Prove that given any relation $R \subseteq X \times X$, its completion to an equivalence relation is the kernel of the coequalizer of the component maps of the relation. For the purpose of this, we define kernels in Set as the equalizer of $f \circ p_1, f \circ p_2 : A \times A \rightarrow B$. (A more general definition of a kernel, independent of zero objects, can be found using a definition of quotients by equivalence relations, and having the kernel be a universal object to factor the quotient through).
4. Prove that if the right arrow in a pullback square is a mono, then so is the left arrow. Thus the intersection as a pullback really is a subobject.
5. Prove that if both the arrows in the pullback 'corner' are mono, then the arrows of the pullback cone are all mono.
6. What is the pullback in the category of posets?
7. What is the pushout in the category of posets?

6. Pairs of constructions

8. Prove that the exponential and the product functors above are adjoints. What are the unit and counit?
9. (worth 4pt) Consider the unique functor $! : C \rightarrow 1$ to the terminal category.
 - a) Does it have a left adjoint? What is it?
 - b) Does it have a right adjoint? What is it?
10. * Prove the propositions in the text.
11. (worth 4pt) Suppose

$$C \begin{array}{c} \xrightarrow{F} \\ \xleftarrow{U} \end{array} D$$

is an adjoint pair. Find a natural transformation $FUF \rightarrow F$. Conclude that there is a natural transformation $\mu : UFUF \rightarrow UF$. Prove that this is associative, in other words that the diagram

$$\begin{array}{ccc} UFUFUF \xrightarrow{UF\mu_X} UFUF & & \\ \mu_{UF} \downarrow & & \downarrow \mu_X \\ UFUF & \xrightarrow{\mu_X} & ufx \end{array}$$

commutes. Prove that the unit of the adjunction forms a unit for this μ , in other words, that the diagram

$$\begin{array}{ccccc} UFX & \xrightarrow{\eta_{UF}} & UFUF & \xleftarrow{UF\eta_X} & UFX \\ & \searrow 1_{UFX} & \downarrow \mu_X & \swarrow 1_{UFX} & \\ & & UFX & & \end{array}$$

commutes.

Monoid objects and monads

Last week we saw what an adjunction was. Here's one thing we can do with adjunctions.

Now, let U be a left adjoint to F . We set $T = UF$. Then we have natural transformations

$$\mu : UFUF \rightarrow UF \quad \mu_X = U\epsilon_{FX}$$

$$\iota : 1 \rightarrow UF \quad \iota_X = \eta_X$$

such that μ is associative and ι is the unit of μ .

These requirements remind us of the definition of a monoid - and this is not that much of a surprise. To see the exact connection, and to garner a wider spread of definitions.

Algebraic objects in categories

We recall the definition of a monoid:

Definition A *monoid* is a set M equipped with an operation $\mu : M \times M \rightarrow M$ that we call *composition* and an operation $e : 1 \rightarrow M$ that we call the identity, such that

- $M \circ 1_M \times M = M \circ M \times 1_M$ (associativity)
- $M \circ 1_M \times e = M \circ e \times 1_M = 1_M$ (unity)

If we have a *monoidal category* - a category C with a bifunctor $\otimes : C \times C \rightarrow C$ called the *tensor product* which is associative (up to natural isomorphisms) and has an object I acting as a unit (up to natural isomorphisms) for the tensor product.

The product in a category certainly works as a tensor product, with a terminal object acting as a unit. However, there is often reason to have a non-commutative tensor product for the monoidal structure of a category. This makes the category a *cartesian monoidal category*.

For, say, abelian groups, or for vector spaces, we have the *tensor product* forming a non-cartesian monoidal category structure. And it is important that we do.

And for the category of endofunctors on a category, we have a monoidal structure induced by composition of endofunctors: $F \otimes G = F \circ G$. The unit is the identity functor.

Now, we can move the definition of a monoid out of the category of sets, and define a generic *monoid object* in a monoidal category:

Definition A *monoid object* in a monoidal category C is an object M equipped with morphisms $\mu : M \otimes M \rightarrow M$ and $e : 1 \rightarrow M$ such that

$$\cdot \quad M \circ 1_M \otimes M = M \circ M \otimes 1_M \text{ (associativity)}$$

$$\cdot \quad M \circ 1_M \otimes e = M \circ e \otimes 1_M = 1_M \text{ (unity)}$$

As an example, a monoid object in the cartesian monoidal category Set is just a monoid. A monoid object in the category of abelian groups is a ring.

A monoid object in the category of abelian groups, with the tensor product for the monoidal structure is a *ring*.

And the composition UF for an adjoint pair is a monoid object in the category of endofunctors on the category.

The same kind of construction can be made translating familiar algebraic definitions into categorical constructions with many different groups of definitions. For *groups*, the corresponding definition introduces a diagonal map $\Delta : G \rightarrow G \times G$, and an inversion map $i : M \rightarrow M$ to codify the entire definition.

One framework that formalizes the whole thing, in such a way that the definitions themselves form a category is the theory of Sketches by Charles Wells. In one formulation we get the following definition:

Definition A *sketch* $S = (G, D, L, K)$ consists of a graph G , a set of diagrams D , a set L of cones in G and a set K of cocones in G .

A *model* of a sketch S in a category C is a graph homomorphism $G \rightarrow C$ such that the image of each diagram in D is commutative, each of the cones is a limit cone and each of the cocones is a colimit cocone.

A *homomorphism of models* is just a natural transformation between the models.

We thus define a *monad in a category* C to be a monoid object in the category of endofunctors on that category.

Specifically, this means:

Definition A *monad* in a category C is an endofunctor $T : C \rightarrow C$ equipped with natural transformations $\mu : T^2 \rightarrow T$ and $\eta : 1 \rightarrow T$ such that the following diagrams commute:

DRAFT: Thursday 2nd February, 2012 16:49

$$\begin{array}{ccccc}
T^3X & \xrightarrow{T\mu_X} & T^2X & & TX & \xrightarrow{\eta_{TX}} & T^2X & \xleftarrow{T\eta_X} & TX \\
\mu_{TX} \downarrow & & \downarrow \mu_X & & \searrow 1_{TX} & & \downarrow \mu_X & & \swarrow 1_{TX} \\
T^2X & \xrightarrow{\mu_X} & TX & & & & TX & &
\end{array}$$

We can take this definition and write it out in Haskell code, as:

```

class Functor m => MathematicalMonad m where
  return :: a -> m a
  join   :: m (m a) -> m a

-- such that
join . fmap return = id           :: m a -> m a
join . return      = id           :: m a -> m a
join . join        = join . fmap join :: m (m (m a)) -> m a

```

Those of you used to Haskell will notice that this is not the same as the `Monad` typeclass. That type class calls for a natural transformation $(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$ (or `bind`).

The secret of the connection between the two lies in the Kleisli category, and a way to build adjunctions out of monads as well as monads out of adjunctions.

Kleisli category

We know that an adjoint pair will give us a monad. But what about getting an adjoint pair out of a monad? Can we reverse the process that got us the monad in the first place?

There are several different ways to do this. Awodey uses the *Eilenberg-Moore category* which has as objects the *algebras of the monad* T : morphisms $h : Tx \rightarrow x$. A morphism $f : (\alpha : TA \rightarrow A) \rightarrow (\beta : TB \rightarrow B)$ is just some morphism $f : A \rightarrow B$ in the category C such that $f \circ \alpha = \beta \circ T(f)$.

We require of T -algebras two additional conditions:

- $1_A = h \circ \eta_A$ (unity)
- $h \circ \mu_A = h \circ Th$ (associativity)

There is a forgetful functor that takes some h to $t(h)$, picking up the object of the T -algebra. Thus $U(h : TA \rightarrow A) = A$, and $U(f) = f$.

We shall construct a left adjoint F to this from the data of the monad T by setting $FC = (\mu_C : T^2C \rightarrow TC)$, making TC the corresponding object. And plugging the corresponding data into the equations, we get:

- $1_{TC} = \mu_C \circ \eta_{TC}$
- $\mu_C \circ \mu_{TC} = \mu_C \circ T\mu_C$

which we recognize as the axioms of unity and associativity for the monad.

$$\begin{array}{ccccc}
 T^3X & \xrightarrow{T\mu_X} & T^2X & & TX & \xrightarrow{\eta_{TX}} & T^2X & \xleftarrow{T\eta_X} & TX \\
 \mu_{TX} \downarrow & & \downarrow \mu_X & & \searrow 1_{TX} & & \downarrow \mu_X & & \swarrow 1_{TX} \\
 T^2X & \xrightarrow{\mu_X} & TX & & & & TX & &
 \end{array}$$

By working through the details of proving this to be an adjunction, and examining the resulting composition, it becomes clear that this is in fact the original monad T . However - while the Eilenberg-Moore construction is highly enlightening for constructing formal systems for algebraic theories, and even for the fixpoint definitions of data types, it is less enlightening to understand Haskell's monad definition.

To get to terms with the Haskell approach, we instead look to a different construction aiming to fulfill the same aim: the *Kleisli category*:

Given a monad T over a category C , equipped with unit η and concatenation μ , we shall construct a new category $K(T)$, and an adjoint pair of functors U, F factorizing the monad into $T = UF$.

We first define $K(T)_0 = C_0$, keeping the objects from the original category.

Then, we set $K(T)_1$ to be the collection of arrows, in C , on the form $A \rightarrow TB$.

The composition of $f : A \rightarrow TB$ with $g : B \rightarrow TC$ is given by the sequence $A \xrightarrow{f} TB \xrightarrow{Tg} T^2C \xrightarrow{\mu_C} TC$

The identity is the arrow $\eta_A : A \rightarrow TA$. The identity property follows directly from the unity axiom for the monad, since η_A composing with μ_A is the identity.

Given this category, we next define the functors:

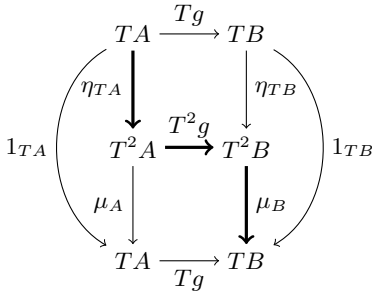
- $U(A) = TA$
- $U(f : A \rightarrow TB) = TA \xrightarrow{Tf} T^2B \xrightarrow{\mu_B} TB$
- $F(A) = A$
- $F(g : A \rightarrow B) = A \xrightarrow{\eta_A} TA \xrightarrow{Tg} TB$

This definition makes U, F an adjoint pair. Furthermore, we get

- $UF(A) = U(A) = TA$

$$\begin{aligned}
 \cdot \quad UF(g : A \rightarrow B) &= U(Tg \circ \eta_A) = \mu_B \circ T(Tg \circ \eta_A) \\
 &= \mu_B \circ T^2 g \circ T\eta_A, \text{ and by naturality of } \mu, \text{ we can rewrite this} \\
 &\text{as} \\
 &= Tg \circ \mu_A \circ T\eta_A = Tg \circ 1_{TA} = Tg \text{ by unitality of } \eta.
 \end{aligned}$$

We've really just chased through this commutative diagram:



Hence, the composite UF really is just the original monad functor T .

But what's the big deal with this? you may ask. The big deal is that we now have a monad specification with a different signature. Indeed, the Kleisli arrow for an arrow $f :: a \rightarrow b$ and a monad `Monad m` is something on the shape $f_k :: a \rightarrow_m b$. And the Kleisli factorization tells us that the Haskell monad specification and the Haskell monad laws are equivalent to their categorical counterparts.

And the composition of Kleisli arrows is easy to write in Haskell:

```

f :: a -> m b
g :: b -> m c

(>=>) :: m a -> (a -> m b) -> m b -- Monadic bind, the Haskell definition

kleisliCompose f g :: a -> m c
kleisliCompose f g = (>=> g) . f

```

Examples

Some monads in Haskell are:

The List monad

Lists form a monad, with the following (redundant) definition:

```

instance Monad [] where
  return x    = [x]

```

```
[] >>= _ = []
(x:xs) >>= f = f x : xs >>= f

join [] = []
join (l:ls) = l ++ join ls
```

As it turns out, the lists monad can be found by considering the free and forgetful functors between sets and monoids. Indeed, the lists are what we get from the Kleene star operation, which is the monad we acquire by composing the free monoid functor with the forgetful functor.

Error handling

We can put a monadic structure on a coproduct $A + B$ so that the monadic bind operation performs computations $A + B \rightarrow A' + B$ until some computation fails, returning an error, typed B , after which we bypass any further computations, just carrying the error out of the entire computation.

The endofunctor here is $+B$. So the monad is given from a way to go from $A + B + B \rightarrow A + B$. Doing this is easy: in Haskell terms, we just remove the constructor differences between the two copies of B floating around. Mathematically, this is just using the functoriality of the coproduct construction on the inclusion maps into $A + B$.

For our example, we shall return the first value of B to ever occur, thus making our join operator look like this:

```
join :: (Either b (Either b a)) -> Either b a
join (Left y) = Left y
join (Right (Left y)) = Left y
join (Right (Right x)) = Right x
```

This gives us a Haskell monad defined by:

```
instance Monad (Either b) where
  return x = Right x

  Left y  >>= _ = Left y
  Right x >>= f = f x
```

Additional reading

- <http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html> (one of the least dramatic monads tutorials out there)

DRAFT: Thursday 2nd February, 2012 16:49

-
- <http://www.disi.unige.it/person/MoggiE/ftp/lc88.ps.gz> (Moggi: *Computational lambda-calculus and monads*, one of the papers that started the interest in monads. Logic, dense reading.)
 - http://www.haskell.org/haskellwiki/Research_papers/Monads_and_arrows (good catalogue over further reading on monads)

Homework

Full marks will be given for 4 out of the 7 questions.

1. Prove that the Kleisli category adjunction is an adjunction.
2. Prove that the Eilenberg-Moore category adjunction is an adjunction.
3. Given monad structures on S and T ,
4. The *writer* monad W is defined by
 - `data Monoid m => W m x = W (x, m)`
 - `fmap f (W (x, m)) = W (f x, m)`
 - `return x = W (x, mempty)`
 - `join (W (W (x, m), n)) = W (x, m `mappend` n)`
 - a) (2pt) Prove that this yields a monad.
 - b) (2pt) Give the Kleisli factorization of the writer monad.
 - c) (2pt) Give the Eilenberg-Moore factorization of the writer monad.
 - d) (2pt) Is there a nice, 'natural' adjunction factorizing the writer monad?

Algebras

Algebras over monads

We recall from the last lecture the definition of an Eilenberg-Moore algebra over a monad $T = (T, \eta, \mu)$:

Definition An *algebra* over a monad T in a category C (a T -algebra) is a morphism $\alpha \in C(TA, A)$, such that the diagrams below both commute:

$$\begin{array}{ccccc}
 A & \xrightarrow{\eta_A} & TA & & T^2A & \xrightarrow{T\alpha} & TA \\
 & \searrow 1_A & \downarrow \alpha & & \downarrow \mu_A & & \downarrow \alpha \\
 & & A & & TA & \xrightarrow{\alpha} & A
 \end{array}$$

While a monad corresponds to the imposition of some structure on the objects in a category, an algebra over that monad corresponds to some evaluation of that structure.

Example: monoids

Let T be the Kleene star monad - the one we get from the adjunction of free and forgetful functors between Monoids and Sets. Then a T -algebra on a set A is equivalent to a monoid structure on A .

Indeed, if we have a monoid structure on A , given by $m : A^2 \rightarrow A$ and $u : 1 \rightarrow A$, we can construct a T -algebra by

$$\begin{aligned}
 \alpha([\]) &= u \\
 \alpha([a_1, a_2, \dots, a_n]) &= m(a_1, \alpha([a_2, \dots, a_n]))
 \end{aligned}$$

This gives us, indeed, a T -algebra structure on A . Associativity and unity follows from the corresponding properties in the monoid.

On the other hand, if we have a T -algebra structure on A , we can construct a monoid structure by setting

$$\begin{aligned}
 u &= \alpha([\]) \\
 m(a, b) &= \alpha([a, b])
 \end{aligned}$$

It is clear that associativity of m follows from the associativity of α , and unitality of u follows from the unitality of α .

Example: Vector spaces

We have free and forgetful functors

$$\mathbf{Set} \xrightarrow{\text{free}} k\text{-Vect} \xrightarrow{\text{forgetful}} \mathbf{Set}$$

forming an adjoint pair; where the free functor takes a set S and returns the vector space with basis S ; while the forgetful functor takes a vector space and returns the set of all its elements.

The composition of these yields a monad T in \mathbf{Set} taking a set S to the set of all formal linear combinations of elements in S . The monad multiplication takes formal linear combinations of formal linear combinations and multiplies them out:

$$3(2v + 5w) - 5(3v + 2w) = 6v + 15w - 15v - 10w = -9v + 5w$$

A T -algebra is a map $\alpha : TA \rightarrow A$ that *acts like a vector space* in the sense that $\alpha(\sum \alpha_i(\sum \beta_j v_j)) = \alpha(\sum \alpha_i \beta_j v_j)$.

We can define $\lambda \cdot v = \alpha(\lambda v)$ and $v + w = \alpha(v + w)$. The operations thus defined are associative, distributive, commutative, and everything else we could wish for in order to define a vector space - precisely because the operations inside TA are, and α is associative.

The moral behind these examples is that using monads and monad algebras, we have significant power in defining and studying algebraic structures with categorical and algebraic tools. This paradigm ties in closely with the theory of *operads* - which has its origins in topology, but has come to good use within certain branches of universal algebra.

An (non-symmetric) *operad* is a graded set $O = \bigoplus_i O_i$ equipped with composition operations $\circ_i : O_n \oplus O_m \rightarrow O_{n+m-1}$ that obey certain unity and associativity conditions. As it turns out, non-symmetric operads correspond to the summands in a monad with polynomial underlying functor, and from a non-symmetric operad we can construct a corresponding monad.

The designator non-symmetric floats in this text to avoid dealing with the slightly more general theory of symmetric operads - which allow us to resort to the input arguments, thus including the symmetrizer of a symmetric monoidal category in the entire definition.

To read more about these correspondences, I can recommend you start with: the blog posts *Monads in Mathematics* here: <http://embuchestissues.wordpress.com/tag/monads-in-mathematics/>

DRAFT: Thursday 2nd February, 2012 16:49

Algebras over endofunctors

Suppose we started out with an endofunctor that is not the underlying functor of a monad - or an endofunctor for which we don't want to settle on a monadic structure. We can still do a lot of the Eilenberg-Moore machinery on this endofunctor - but we don't get quite the power of algebraic specification that monads offer us. At the core, here, lies the lack of associativity for a generic endofunctor - and algebras over endofunctors, once defined, will correspond to non-associative correspondences to their monadic counterparts.

Definition For an endofunctor $P : C \rightarrow C$, we define a P -algebra to be an arrow $\alpha \in C(PA, A)$.

A homomorphism of P -algebras $\alpha \rightarrow \beta$ is some arrow $f : A \rightarrow B$ such that the diagram below commutes:

$$\begin{array}{ccc} PA & \xrightarrow{Pf} & PB \\ \alpha \downarrow & & \downarrow \beta \\ A & \xrightarrow{f} & B \end{array}$$

This homomorphism definition does not need much work to apply to the monadic case as well.

Example: Groups

A group is a set G with operations $u : 1 \rightarrow G, i : G \rightarrow G, m : G \times G \rightarrow G$, such that u is a unit for m , m is associative, and i is an inverse.

Ignoring for a moment the properties, the theory of groups is captured by these three maps, or by a diagram

$$\begin{array}{ccc} & 1 & \\ & \downarrow u & \\ G \times G & \xrightarrow{m} G \xleftarrow{i} & G \end{array}$$

We can summarize the diagram as $1 + G + G \times G \mapsto^{[u, i, m]} G$ and thus recognize that groups are some equationally defined subcategory of the category of T -algebras for the polynomial functor $T(X) = 1 + X + X \times X$. The subcategory is *full*, since if we have two algebras $\gamma : T(G) \rightarrow G$ and $\eta : T(H) \rightarrow H$, that both lie within the subcategory that fulfills all the additional axioms, then certainly any morphism $\gamma \rightarrow \eta$ will be compatible with the structure maps, and thus will be a group homomorphism.

We shall denote the category of P -algebras in a category C by $P\text{-Alg}(C)$, or just $P\text{-Alg}$ if the category is implicitly understood.

This category is wider than the corresponding concept for a monad. We don't require the kind of associativity we would for a monad - we just lock down the underlying structure. This distinction is best understood with an example:

The free monoids monad has monoids for its algebras. On the other hand, we can pick out the underlying functor of that monad, forgetting about the unit and multiplication. An algebra over this structure is a slightly more general object: we no longer require $(a \cdot b) \cdot c = a \cdot (b \cdot c)$, and thus, the theory we get is that of a *magma*. We have concatenation, but we can't drop the brackets, and so we get something more reminiscent of a binary tree.

Initial P -algebras and recursion

Consider the polynomial functor $P(X) = 1 + X$ on the category of sets. Its algebras form a category, by the definitions above - and an algebra needs to pick out one special element 0, and one endomorphism T , for a given set.

What would an initial object in this category of P -algebras look like? It would be an object I equipped with maps $1 \rightarrow^o I \leftarrow^n I$. For any other pair of maps $a : 1 \rightarrow X, s : X \rightarrow X$, we'd have a unique arrow $u : I \rightarrow X$ such that

$$\begin{array}{ccc} 1 + I & \xrightarrow{1_1 + u} & 1 + X \\ \downarrow [o, n] & & \downarrow [a, s] \\ I & \xrightarrow{u} & X \end{array}$$

commutes, or in equations such that $u(o) = a$ and $u(n(x)) = s(u(x))$.

Now, unwrapping the definitions in place, we notice that we will have elements $o, n(o), n(n(o)), \dots$ in I , and the initiality will force us to not have any *other* elements floating around. Also, initiality will prevent us from having any elements not in this minimally forced list.

We can rename the elements to form something more recognizable - by equating an element in I with the number of applications of n to o . This yields, for us, elements $0, 1, 2, \dots$ with one function that picks out the 0, and another that gives us the successor.

DRAFT: Thursday 2nd February, 2012 16:49

This should be recognizable as exactly the natural numbers; with just enough structure on them to make the principle of mathematical induction work: suppose we can prove some statement $P(0)$, and we can extend a proof of $P(n)$ to $P(n+1)$. Then induction tells us that the statement holds for all $P(n)$.

More importantly, recursive definitions of functions from natural numbers can be performed here by choosing an appropriate algebra mapping to.

This correspondence between the initial object of $P(X) = 1 + X$ is the reason such an initial object in a category with coproducts and terminal objects is called a *natural numbers object*.

For another example, we consider the functor $P(X) = 1 + X \times X$.

Pop Quiz Can you think of a structure with this as underlying defining functor?

An initial $1 + X \times X$ -algebra would be some diagram

$$1 \rightarrow^o I \leftarrow^m I \times I$$

such that for any other such diagram

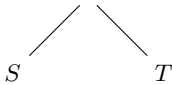
$$1 \rightarrow^a X \leftarrow^* X \times X$$

we have a unique arrow $u : I \rightarrow X$ such that

$$\begin{array}{ccc} 1 + I \times I & \xrightarrow{1_1 + u \times u} & 1 + X \times X \\ \downarrow [o, n] & & \downarrow [a, s] \\ I & \xrightarrow{u} & X \end{array}$$

commutes.

Unwrapping the definition, working over Sets again, we find we are forced to have some element $*$, the image of o . Any two elements S, T in the set give rise to some (S, T) , which we can view as being the binary tree



The same way that we could construct induction as an algebra map from a natural numbers object, we can use this object to construct a tree-shaped induction; and similarly, we can develop what amounts to the theory of *structural induction* using these more general approaches to induction.

Example of structural induction

Using the structure of $1 + X \times X$ -algebras we shall prove the following statement:

Proposition The number of leaves in a binary tree is one more than the number of internal nodes.

Proof We write down the actual Haskell data type for the binary tree initial algebra.

```
data Tree = Leaf | Node Tree Tree

nLeaves Leaf = 1
nLeaves (Node s t) = nLeaves s + nLeaves t

nNodes Leaf = 0
nNodes (Node s t) = 1 + nNodes s + nNodes t
```

Now, it is clear, as a base case, that for the no-nodes tree Leaf:

```
nLeaves Leaf = 1 + nNodes Leaf
```

For the structural induction, now, we consider some binary tree, where we assume the statement to be known for each of the two subtrees. Hence, we have

```
tree = Node s t

nLeaves s = 1 + nNodes s
nLeaves t = 1 + nNodes t

and we may compute

nLeaves tree = nLeaves s + nLeaves t
              = 1 + nNodes s + 1 + nNodes t
              = 2 + nNodes s + nNodes t

nNodes tree = 1 + nNodes s + nNodes t
```

Now, since the statement is proven for each of the cases in the structural description of the data, it follows from the principle of structural induction that the proof is finished.

In order to really nail down what we are doing here, we need to define what we mean by predicates in a strict manner. There is a way to do this using *fibrations*, but this reaches far outside the scope of this course. For the really interested reader, I'll refer to <http://www.springerlink.com/content/d022nlv03n26nm03/>.

Another way to do this is to introduce a *topos*, and work it all out in terms of its *internal logic*, but again, this reaches outside the scope of this course.

DRAFT: Thursday 2nd February, 2012 16:49

Lambek's lemma

What we do when we write a recursive data type definition in Haskell *really* to some extent is to define a data type as the initial algebra of the corresponding functor. This intuitive equivalence is vindicated by the following

Lemma Lambek If $P : C \rightarrow C$ has an initial algebra I , then $P(I) = I$.

Proof Let $a : PA \rightarrow A$ be an initial P -algebra. We can apply P again, and get a chain $PPA \rightarrow^{Pa} PA \xrightarrow{a} A$.

We can fill out the diagram

$$PA \xrightarrow{\alpha} A$$

$$PPA \xrightarrow{P\alpha} PA \xrightarrow{\alpha} A$$

to form the diagram

$$\begin{array}{ccccc} & PA & \xrightarrow{\alpha} & A & \\ & \swarrow Ff & & \searrow = & \\ PPA & \xrightarrow{P\alpha} & PA & \xrightarrow{\alpha} & A \\ & \nwarrow & \swarrow f & & \searrow = \end{array}$$

where f is induced by initiality, since $Pa : PPA \rightarrow PA$ is also a P -algebra.

The diagram above commutes, and thus $af = 1_{PA}$ and $fa = 1_A$. Thus f is an inverse to a . QED.

Thus, by Lambek's lemma we *know* that if $P_A(X) = 1 + A \times X$ then for that P_A , the initial algebra - should it exist - will fulfill $I = 1 + A \times I$, which in turn is exactly what we write, defining this, in Haskell code:

```
List a = Nil | Cons a List
```

Recursive definitions with the unique maps from the initial algebra

Consider the following $P_A(X)$ -algebra structure $l : P_A(\mathbb{N}) \rightarrow \mathbb{N}$ on the natural numbers:

$$l(*) = 0$$

$$l(a, n) = 1 + n$$

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

We get a unique map f from the initial algebra for $P_A(X)$ (lists of elements of type A) to \mathbb{N} from this definition. This map will fulfill:

```
f(Nil) = 1(*) = 0
f(Cons a xs) = 1(a, f(xs)) = 1 + f(xs)
```

which starts taking on the shape of the usual definition of the length of a list:

```
length(Nil) = 0
length(Cons a xs) = 1 + length(xs)
```

And thus, the machinery of endofunctor algebras gives us a strong method for doing recursive definitions in a theoretically sound manner.

Homework

Complete credit will be given for 8 of the 13 questions.

1. Find a monad whose algebras are *associative algebras*: vector spaces with a binary, associative, unitary operation (multiplication) defined on them. Factorize the monad into a free/forgetful adjoint pair.
2. Find an endofunctor of Hask whose initial object describes trees that are either binary or ternary at each point, carrying values from some A in the leaves.
3. Write an implementation of the monad of vector spaces in Haskell. If this is tricky, restrict the domain of the monad to, say, a 3-element set, and implement the specific example of a 3-dimensional vector space as a monad. Hint: <http://sigfpe.blogspot.com> has written about this approach.
4. Find a $X \mapsto 1 + A \times X$ -algebra L such that the unique map from the initial algebra I to L results in the function that will reverse a given list.
5. Find a $X \mapsto 1 + A \times X$ -algebra structure on the object $1 + A$ that will pick out the first element of a list, if possible.
6. Find a $X \mapsto \mathbb{N} + X \times X$ -algebra structure on the object \mathbb{N} that will pick out the sum of the leaf values for the binary tree in the initial object.
7. Complete the proof of Lambek's lemma by proving the diagram commutes.

DRAFT: Thursday 2nd February, 2012 16:49

-
8. * We define a *coalgebra* for an endofunctor T to be some arrow $\gamma : A \rightarrow TA$. If T is a comonad - i.e. equipped with a counit $\epsilon : T \rightarrow 1$ and a cocomposition $\Delta : T \rightarrow T \times T$, then we define a coalgebra for the comonad T to additionally fulfill $\gamma \circ T\gamma = \gamma \circ \Delta$ (compatibility) and $\epsilon_A \circ \gamma = 1_A$ (counitality).
- a) (2pt) Prove that if T is an endofunctor, then if T has an initial algebra, then it is a coalgebra. Does T necessarily have a final coalgebra?
 - b) (2pt) Prove that if U, F are an adjoint pair, then FU forms a comonad.
 - c) (2pt) Describe a final coalgebra over the comonad formed from the free/forgetful adjunction between the categories of Monoids and Sets.
 - d) (2pt) Describe a final coalgebra over the endofunctor $P(X) = 1 + X$.
 - e) (2pt) Describe a final coalgebra over the endofunctor $P(X) = 1 + A \times X$.
 - f) (2pt) Prove that if $c : C \rightarrow PC$ is a final coalgebra for an endofunctor $P : C \rightarrow C$, then c is an isomorphism.

The morphism zoo

Recursion patterns

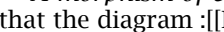
Meijer, Fokkinga & Patterson identified in the paper *Functional programming with bananas, lenses, envelopes and barbed wire* a number of generic patterns for recursive programming that they had observed, catalogued and systematized. The aim of that paper is to establish a number of *rules* for modifying and rewriting expressions involving these generic recursion patterns.

As it turns out, these patterns are instances of the same phenomenon we saw last lecture: where the recursion comes from specifying a different algebra, and then take a uniquely existing morphism induced by initiality (or, as we shall see, finality).

Before we go through the recursion patterns, we need to establish a few pieces of theoretical language, dualizing the Eilenberg-Moore algebra constructions from the last lecture.

Coalgebras for endofunctors

Definition If $P : C \rightarrow C$ is an endofunctor, then a P -coalgebra on A is a morphism $a : A \rightarrow PA$.

A *morphism of coalgebras*: $f : a \rightarrow b$ is some $f : A \rightarrow B$ such that the diagram  commutes.

Just as with algebras, we get a category of coalgebras. And the interesting objects here are the *final coalgebras*. Just as with algebras, we have

Lemma (Lambek) If $a : A \rightarrow PA$ is a final coalgebra, it is an isomorphism.

Finally, one thing that makes us care highly about these entities: in an appropriate category (such as ω -CPO), initial algebras and final coalgebras coincide, with the correspondence given by inverting the algebra/coalgebra morphism. In Haskell not quite true (specifically, the final coalgebra for the lists functor gives us streams...).

Onwards to recursion schemes!

We shall define a few specific morphisms we'll use repeatedly. This notation, introduced here, occurs all over the place in these corners of the literature, and are good to be aware of in general:

- If $a : TA \rightarrow A$ is an initial algebra for T , we denote $a = in_A$.
- If $a : A \rightarrow TA$ is a final coalgebra for T , we denote $a = out_A$.
- We write μf for the fixed point operator

$\mu f = x \text{ where } x = f x$

We note that in the situation considered by MFP, initial algebras and final coalgebras coincide, and thus in_A, out_A are the pair of isomorphic maps induced by either the initial algebra- or the final coalgebra-structure.

Catamorphisms

A *catamorphism* is the uniquely existing morphism from an initial algebra to a different algebra. We have to define maps down to the return value type for each of the constructors of the complex data type we're recursing over, and the catamorphism will deconstruct the structure (trees, lists, ...) and do a generalized *fold* over the structure at hand before returning the final value.

The intuition is that for catamorphisms we start essentially structured, and dismantle the structure.

Example: the length function from last lecture. This is the catamorphism for the functor $P_A(X) = 1 + A \times X$ given by the maps

```
u :: Int
u = 0

m :: (A, Int) -> Int
m (a, n) = n+1
```

MFP define the catamorphism by, supposing T is initial for the functor F :

```
cata :: (F a b -> b) -> T a -> b
cata phi = mu (\x -> phi . fmap x . outT)
```

We can reframe the example above as a catamorphism by observing that here,

DRAFT: Thursday 2nd February, 2012 16:49

```
data F a b = Nil | Cons a b deriving (Eq, Show)
type T a = [a]
```

```
instance Functor (F a) where
  fmap _ Nil = Nil
  fmap f (Cons n a) = Cons n (f a)
```

```
outT :: T a -> F a (T a)
outT [] = Nil
outT (a:as) = Cons a as
```

```
lphi :: F a Int -> Int
lphi Nil = 0
lphi (Cons a n) = n + 1
```

```
l = cata lphi
```

where we observe that μ has a global definition for everything we do and out is defined once we settle on the functor F and its initial algebra. Thus, the definition of phi really is the only place that the recursion data shows up.

Anamorphisms

An *anamorphism* is the categorical dual to the catamorphism. It is the canonical morphism from a coalgebra to the final coalgebra for that endofunctor.

Here, we start unstructured, and erect a structure, induced by the coalgebra structures involved.

Example: we can write a recursive function

```
first :: Int -> [Int]
first 1 = [1]
first n = n : first (n - 1)
```

This is an anamorphism from the coalgebra for $P_{\mathbb{N}}(X) = 1 + \mathbb{N} \times X$ on \mathbb{N} generated by the two maps

```
c 0 = Left ()
c n = Right (n, n-1)
```

and we observe that we can chase through the diagram :[[Image:CoalgebraMorphism.png]] to conclude that therefore

```
f 0 = []
f n = n : f (n - 1)
```

which is exactly the recursion we wrote to begin with.

MFP define the anamorphism by a fixpoint as well, namely:

DRAFT: Thursday 2nd February, 2012 16:49
Please do not circulate

```
ana :: (b -> F a b) -> b -> T a
ana psi = mu (\x -> inT . fmap x . psi)
```

We can, again, recast our illustration above into a structural anamorphism, by:

```
-- Reuse mu, F, T from above
inT :: F a (T a) -> T a
inT Nil = []
inT (Cons a as) = a:as

fpsi :: Int -> F Int Int
fpsi 0 = Nil
fpsi n = Cons n (n-1)
```

Again, we can note that the implementation of `fpsi` here is exactly the `c` above, and the resulting function will - as we can verify by compiling and running - give us the same kind of reversed list of the `n` first integers as the `first` function above would.

Hylomorphisms

The *hylomorphisms* capture one of the two possible compositions of anamorphisms and catamorphisms. Parametrized over an algebra $\phi : TA \rightarrow A$ and a coalgebra $\psi : B \rightarrow TB$ the hylomorphism is a recursion pattern that computes a value in A from a value in A by generating some sort of intermediate structure and then collapsing it again.

It is, thus the composition of the uniquely existing morphism from a coalgebra to the final coalgebra for an endofunctor, followed by the uniquely existing morphism from the initial algebra to some other algebra.

MFP define it, again, as a fix point:

```
hylo :: (F a b2 -> b2) -> (b1 -> F a b1) -> b1 -> b2
hylo phi psi = mu (\x -> phi . fmap x . psi)
```

First off, we can observe that by picking one or the other of in_A, out_A as a parameter, we can recover both the anamorphisms and the catamorphisms as hylomorphisms.

As an example, we'll compute the factorial function using a hylomorphism:

```
phi :: F Int Int -> Int
phi Nil = 1
phi (Cons n m) = n*m

psi :: Int -> F Int Int
```

DRAFT: Thursday 2nd February, 2012 16:49

```
psi 0 = Int
psi n = Cons n (n-1)

factorial = hylo phi psi
```

Metamorphisms

The *metamorphism* is the *other* composition of an anamorphism with a catamorphism. It takes some structure, deconstructs it, and then reconstructs a new structure from it.

As a recursion pattern, it's kinda boring - it'll take an interesting structure, deconstruct it into a *scalar* value, and then reconstruct some structure from that scalar. As such, it won't even capture the richness of $\text{hom}(Fx, Gy)$, since any morphism expressed as a metamorphism will factor through a map $x \rightarrow y$.

Paramorphisms

Paramorphisms were discussed in the MFP paper as a way to extend the catamorphisms so that the operating function can access its arguments in computation as well as in recursion. We gave the factorial above as a hylomorphism instead of a catamorphism precisely because no simple enough catamorphic structure exists.

Apomorphisms

The *apomorphism* is the dual of the paramorphism - it does with retention of values along the way what anamorphisms do compared to catamorphisms.

Further reading

- Erik Meijer, Maarten Fokkinga, Ross Paterson: *Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire* <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.125>
- L. Augusteijn: *Sorting morphisms* <http://www.springerlink.com/index/w543447353067h67.pdf>

Further properties of adjunctions

RAPL

Proposition If F is a *right adjoint*, thus if F has a left adjoint, then F preserves limits in the sense that $F(\lim_{\leftarrow} A_i) = \lim_{\leftarrow} F(A_i)$.

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

Example: $(\lim_{\leftarrow i} A_i) \times X = \lim_{\leftarrow i} A_i \times X$.

We can use this to prove that things cannot be adjoints - since all right adjoints preserve limits, if a functor G doesn't preserve limits, then it doesn't have a left adjoint.

Similarly, and dually, left adjoints preserve colimits. Thus if a functor doesn't preserve colimits, it cannot be a left adjoint, thus cannot have a right adjoint.

The proof of these statements build on the *Yoneda lemma*:

Lemma If C is a locally small category (i.e. all hom-sets are sets). Then for any $c \in C_0$ and any functor $F : C^{op} \rightarrow Sets$ there is an isomorphism $hom_{hom_{Sets} C^{op}}(yC, F) = FC$ where we define $yC = d \mapsto hom_C(d, c) : C^{op} \rightarrow Sets$.

The Yoneda lemma has one important corollary:

Corollary If $yA = yB$ then $A = B$.

Which, in turn has a number of important corollaries:

Corollary $(A^B)^C = A^{B \times C}$

Corollary Adjoints are unique up to isomorphism - in particular, if $F : C \rightarrow D$ is a functor with right adjoints $U, V : D \rightarrow C$, then $U = V$.

Proof $hom_C(C, UD) = hom_D(FC, D) = hom_C(C, VD)$, and thus by the corollary to the Yoneda lemma, $UD = VD$, natural in D .

Functors that are adjoints

- The functor $X \mapsto X \times A$ has right adjoint $Y \mapsto Y^A$. The universal mapping property of the exponentials follows from the adjointness property.
- The functor $\Delta : C \rightarrow C \times C, c \mapsto (c, c)$ has a left adjoint given by the coproduct $(X, Y) \mapsto X + Y$ and right adjoint the product $(X, Y) \mapsto X \times Y$.
- More generally, the functor $C \rightarrow C^J$ that takes c to the constant functor $const_c(j) = c, const_c(f) = 1_c$ has left and right adjoints given by colimits and limits:

$$\lim_{\rightarrow} - \mid \Delta - \mid \lim_{\leftarrow}$$

- Pointed rings are pairs $(R, r \in R)$ of rings and one element singled out for attention. Homomorphisms of pointed rings need to take the distinguished point to the distinguished point. There is an obvious forgetful functor $U : Rings_* \rightarrow Rings$, and this has a left adjoint - a free ring functor that adjoins a new indeterminate $R \mapsto (R[x], x)$. This gives a formal definition of what we mean by *formal polynomial expressions* et.c.

- Given sets A, B , we can consider the powersets $P(A), P(B)$ containing, as elements, all subsets of A, B respectively. Suppose $f : A \rightarrow B$ is a function, then $f^{-1} : P(B) \rightarrow P(A)$ takes subsets of B to subsets of A .

Viewing $P(A)$ and $P(B)$ as partially ordered sets by the inclusion operations, and then as categories induced by the partial order, f^{-1} turns into a functor between partial orders. And it turns out f^{-1} has a left adjoint given by the operation $im(f)$ taking a subset to the set of images under the function f . And it has a right adjoint $f_*(U) = \{b \in B : f^{-1}(b) \subseteq U\}$

- We can introduce a categorical structure to logic. We let L be a formal language, say of predicate logic. Then for any list $x = x_1, x_2, \dots, x_n$ of variables, we have a preorder $Form(x)$ of formulas with no free variables not occurring in x . The preorder on $Form(x)$ comes from the *entailment* operation - $f| - g$ if in every interpretation of the language, $f \Rightarrow g$.

We can build an operation on these preorders - a functor on the underlying categories - by adjoining a single new variable: $*$: $Form(x) \rightarrow Form(x, y)$, sending each form to itself. Obviously, if $f| - g$ with x the source of free variables, if we introduce a new allowable free variable, but don't actually change the formulas, the entailment stays the same.

It turns out that there is a right adjoint to $*$ given by $f \mapsto \forall y.f$. And a left adjoint to $*$ given by $f \mapsto \exists y.f$. Adjoints properties give us classical deduction rules from logic.

Homework

1. Write a fold for the data type `data T a = L a | B a a | C a a a` and demonstrate how this can be written as a catamorphism by giving the algebra it maps to.
2. Write the fibonacci function as a hylomorphism.
3. Write the Towers of Hanoi as a hylomorphism. You'll probably want to use binary trees as the intermediate data structure.
4. Write a prime numbers generator as an anamorphism.
5. * The integers have a partial order induced by the divisibility relation. We can thus take any integer and arrange all its divisors in a tree by having an edge $n \rightarrow d$ if $d|n$ and d

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

doesn't divide any other divisor of n . Write an anamorphic function that will generate this tree for a given starting integer. Demonstrate how this function is an anamorphism by giving the algebra it maps from.

Hint: You will be helped by having a function to generate a list of all primes. One suggestion is:

```
primes :: [Integer]
primes = sieve [2..]
  where
    sieve (p:xs) = p : sieve [x|x <- xs, x `mod` p > 0]
```

Hint: A good data structure to use is; with expected output of running the algorithm:

```
data Tree = Leaf Integer | Node Integer [Tree]

divisionTree 60 =
  Node 60 [
    Node 30 [
      Node 15 [
        Leaf 5,
        Leaf 3],
      Node 10 [
        Leaf 5,
        Leaf 2],
      Node 6 [
        Leaf 3,
        Leaf 2]],
    Node 20 [
      Node 10 [
        Leaf 5,
        Leaf 2],
      Node 4 [
        Leaf 2]],
    Node 12 [
      Node 6 [
        Leaf 3,
        Leaf 2],
      Node 4 [
        Leaf 2]]]
```

Topoi

This lecture will be shallow, and leave many things undefined, hinted at, and is mostly meant as an appetizer, enticing the audience to go forth and seek out the literature on topos theory for further studies.

Subobject classifier

One very useful property of the category \mathbf{Set} is that the powerset of a given set is still a set; we have an internal concept of *object of all subobjects*. Certainly, for any category (small enough) \mathcal{C} , we have a contravariant functor $\text{Sub}(-) : \mathcal{C} \rightarrow \mathbf{Set}$ taking an object to the set of all equivalence classes of monomorphisms into that object; with the image $\text{Sub}(f)$ given by the pullback diagram

$$\begin{array}{ccc} \text{Sub}(f)W & \longrightarrow & W \\ \text{Sub}(f)(w) \downarrow & & \downarrow w \\ A & \xrightarrow{f} & B \end{array}$$

If the functor $\text{Sub}(-)$ is *representable* - meaning that there is some object $X \in \mathcal{C}_0$ such that $\text{Sub}(-) = \text{hom}(-, X)$ - then the theory surrounding representable functors, connected to the Yoneda lemma - give us a number of good properties.

One of them is that every representable functor has a *universal element*; a generalization of the kind of universal mapping properties we've seen in definitions over and over again during this course; all the definitions that posit the unique existence of some arrow in some diagram given all other arrows.

Thus, in a category with a representable subobject functor, we can pick a representing object $\Omega \in \mathcal{C}_0$, such that $\text{Sub}(X) = \text{hom}(X, \Omega)$. Furthermore, picking a universal element corresponds to picking a subobject $\Omega_0 \hookrightarrow \Omega$ such that for any object A and sub-

object $A_0 \hookrightarrow A$, there is a unique arrow $\chi : A \rightarrow \Omega$ such that there is a pullback diagram

$$\begin{array}{ccc} A_0 & \xrightarrow{\quad} & \Omega_0 \\ \downarrow & & \downarrow \\ A & \xrightarrow{\chi} & \Omega \end{array}$$

One can prove that Ω_0 is terminal in \mathcal{C} , and we shall call Ω the *subobject classifier*, and this arrow $\Omega_0 = 1 \rightarrow \Omega$ *true*. The arrow χ is called the characteristic arrow of the subobject.

In Set, all this takes on a familiar tone: the subobject classifier is a 2-element set, with a *true* element distinguished; and a characteristic function of a subset takes on the *true* value for every element in the subset, and the other (false) value for every element not in the subset.

Defining topoi

Definition A *topos* is a cartesian closed category with all finite limits and with a subobject classifier.

It is worth noting that this is a far stronger condition than anything we can even hope to fulfill for the category of Haskell types and functions. The functional programming relevance will take a back seat in this lecture, in favour of usefulness in logic and set theory replacements.

Properties of topoi

The meat is in the properties we can prove about topoi, and in the things that turn out to be topoi.

Theorem Let \mathcal{E} be a topos. \mathcal{E} has finite colimits.

Power object

Since a topos is closed, we can take exponentials. Specifically, we can consider $[A \rightarrow \Omega]$. This is an object such that $\text{hom}(B, [A \rightarrow \Omega]) = \text{hom}(A \times B, \Omega) = \text{Sub}(A \times B)$. Hence, we get an internal version of the subobject functor. (pick B to be the terminal object to get a sense for how global elements of $[A \rightarrow \Omega]$ correspond to subobjects of A)

Internal logic

We can use the properties of a topos to develop a logic theory - mimicking the development of logic by considering operations on subsets in a given universe:

Classically, in Set, and predicate logic, we would say that a *predicate* is some function from a universe to a set of truth values. So a predicate takes some sort of objects, and returns either True or False.

Furthermore, we allow the definition of sets using predicates:

$$\{x \in U : P(x)\}$$

Looking back, though, there is no essential difference between this, and defining the predicate as the subset of the universe directly; the predicate-as-function appears, then, as the characteristic function of the subset. And types are added as easily - we specify each variable, each object, to have a set it belongs to.

This way, predicates really are subsets. Type annotations decide which set the predicate lives in. And we have everything set up in a way that opens up for the topos language above.

We'd define, for predicates P, Q acting on the same type:

- $\{x \in A : \top\} = A$
- $\{x \in A : \perp\} = \emptyset$
- $\{x : (P \wedge Q)(x)\} = \{x : P(x)\} \cap \{x : Q(x)\}$
- $\{x : (P \vee Q)(x)\} = \{x : P(x)\} \cup \{x : Q(x)\}$
- $\{x \in A : (\neg P)(x)\} = A \setminus \{x \in A : P(x)\}$

We could then start to define primitive logic connectives as set operations; the intersection of two sets is the set on which **both** the corresponding predicates hold true, so $\wedge = \cap$. Similarly, the union of two sets is the set on which either of the corresponding predicates holds true, so $\vee = \cup$. The complement of a set, in the universe, is the negation of the predicate, and all other propositional connectives (implication, equivalence, ...) can be built with conjunction (and), disjunction (or) and negation (not).

So we can mimic all these in a given topos:

We say that a *universe* U is just an object in a given topos. (Note that by admitting several universes, we arrive at a *typed* predicate logic, with basically no extra work.)

A *predicate* is a subobject of the universe.

We can now proceed to define all the familiar logic connectives one by one, using the topos setting. While doing this, we shall introduce the notation $t_A : A \rightarrow \Omega$ for the morphism $t_A =$

$A \rightarrow 1 \xrightarrow{true} \Omega$ that takes on the value *true* on all of A . We note that with this convention, χ_{A_0} , the characteristic morphism of a subobject, is the arrow such that $\chi_{A_0} \circ i = t_{A_0}$.

Conjunction: Given predicates P, Q , we need to define the *conjunction* $P \wedge Q$ as some $P \wedge Q : U \rightarrow \Omega$ that corresponds to both P and Q simultaneously.

We may define $true \times true : 1 \rightarrow \Omega \times \Omega$, a subobject of $\Omega \times \Omega$. Being a subobject, this has a characteristic arrow $\wedge : \Omega \times \Omega \rightarrow \Omega$, that we call the *conjunction arrow*.

Now, we may define $\chi_P \times \chi_Q : U \rightarrow \Omega \times \Omega$ for subobjects $P, Q \subseteq U$ - and we take their composition $\wedge \circ \chi_P \times \chi_Q$ to be the characteristic arrow of the subobject $P \wedge Q$.

And, indeed, this results in a topoidal version of intersection of subobjects.

Implication: Next we define $\leq_1 : \Omega_1 \rightarrow \Omega \times \Omega$ to be the equalizer of \wedge and $proj_1$. Given $v, w : U \rightarrow \Omega$, we write $v \leq_1 w$ if $v \times w$ factors through \leq_1 .

Using the definition of an equalizer we arrive at $v \leq_1 w$ iff $v = v \wedge w$. From this, we can deduce

- $u \leq_1 true$
- $u \leq_1 u$
- If $u \leq_1 v$ and $v \leq_1 w$ then $u \leq_1 w$.
- If $u \leq_1 v$ and $v \leq_1 u$ then $u = v$ and thus, \leq_1 is a partial order on $[U \rightarrow \Omega]$. Intuitively, $u \leq_1 v$ if v is at least as true as u .

This relation corresponds to inclusion on subobjects. Note that $\leq_1 : \Omega_1 \rightarrow \Omega \times \Omega$, given from the equalizer, gives us Ω_1 as a *relation* on Ω - a subobject of $\Omega \times \Omega$. Specifically, it has a classifying arrow $\Rightarrow : \Omega \times \Omega \rightarrow \Omega$. We write $h \Rightarrow k = (\Rightarrow) \circ h \times k$. And for subobjects $P, Q \subseteq A$, we write $P \Rightarrow Q$ for the subobject classified by $\chi_P \Rightarrow \chi_Q$.

It turns out, without much work, that this $P \Rightarrow Q$ behaves just like classical implication in its relationship to \wedge .

Membership: We can internalize the notion of *membership* as a subobject $\in^U \subseteq U \times \Omega^U$, and thus get the membership relation from a pullback:

DRAFT: Thursday 2nd February, 2012 16:49

$$\begin{array}{ccc}
\in^U & \xrightarrow{\quad} & 1 \\
\downarrow & & \downarrow \\
U \times \Omega^U & \xrightarrow{\quad} & \Omega^U \times U \xrightarrow{\text{ev}_U} \Omega
\end{array}$$

For elements $x \times h : 1 \rightarrow U \times \Omega^U$, we write $x \in^U h$ for $x \times h \in \in^U$. Yielding a subset of the product $U \times \Omega^U$, this is readily interpretable as a relation relating things in U with subsets of U , so that for any x, h we can answer whether $x \in^U h$. Both notations indicate $\text{ev}_A \circ h \times x = \text{true}$.

Universal quantification: For any object U , the maximal subobject of U is U itself, embedded with 1_U into itself. There is an arrow $\tau_U : 1 \rightarrow \Omega^U$ represents this subobject. Being a map from 1 , it is specifically monic, so it has a classifying arrow $\forall_U : \Omega^U \rightarrow \Omega$ that takes a given subobject of U to *true* precisely if it is in fact the maximal subobject.

Now, with a relation $r : R \rightarrow B \times A$, we define $\forall a.R$ by the following pullback:

$$\begin{array}{ccc}
\forall a.R & \xrightarrow{\quad} & 1 \\
\downarrow \forall a.r & & \downarrow \tau_A \\
B & \xrightarrow{\lambda_{\chi_R}} & \Omega^A
\end{array}$$

where λ_{χ_R} comes from the universal property of the exponential.

Theorem For any $s : S \rightarrow B$ monic, $S \subseteq \forall a.R$ iff $S \times A \subseteq R$.

This theorem tells us that the subobject given by $\forall a.R$ is the largest subobject of B that is related by R to all of A .

Falsum: We can define the *false* truth value using these tools as $\forall w \in \Omega.w$. This might be familiar to the more advanced Haskell type hackers - as the type

`x :: forall a. a`

which has to be able to give us an element of any type, regardless of the type itself. And in Haskell, the only element that inhabits all types is `undefined`.

From a logical perspective, we use a few basic inference rules:

$$\begin{array}{ccc}
\frac{*}{\phi : \phi} & \frac{\Gamma : \phi}{\Gamma : \forall x.\phi} & \frac{\Gamma : \forall x.\phi}{\Gamma : \phi} \\
& & \frac{\Gamma : \phi}{\Gamma(x/s) : \phi(x/s)}
\end{array}$$

and connect them up to derive

$$\frac{\frac{*}{\forall w.w:\forall w.w}}{\forall w.w:w}$$

for any ϕ not involving w - and we can always adjust any ϕ to avoid w .

Thus, the formula $\forall w.w$ has the property that it implies everything - and thus is a good candidate for the *false* truth value; since the inference

$$\frac{*}{\perp : \phi}$$

is the defining introduction rule for false.

Negation: We define negation the same way as in classical logic: $\neg\phi = \phi \Rightarrow \text{false}$.

Disjunction: We can define

$$P \vee Q = \forall w.((\phi \Rightarrow w) \wedge (\psi \Rightarrow w)) \Rightarrow w$$

Note that this definition uses one of our primary inference rules:

$$\frac{\Gamma, \phi : \theta \quad \Gamma, \psi : \theta}{\Gamma, \phi \vee \psi : \theta} \quad \frac{\Gamma, \phi \vee \psi : \theta}{\Gamma, \phi : \theta \quad \Gamma, \psi : \theta}$$

as the defining property for the disjunction, and we may derive any properties we like from these.

Existential quantifier: Finally, the existential quantifier is derived similarly to the disjunction - by figuring out a rule we want it to obey, and using that as a definition for it:

$$\exists x.\phi = \forall w.(\forall x.\phi \Rightarrow w) \Rightarrow w$$

Here, the rule we use as defining property is

$$\frac{\Gamma, \phi : \psi}{\Gamma, \exists x.\phi : \psi} \quad \frac{\Gamma, \exists x.\phi : \psi}{\Gamma, \phi : \psi}$$

Before we leave this exploration of logic, some properties worth knowing about: While we can prove $\neg(\phi \wedge \neg\phi)$ and $\phi \Rightarrow \neg\neg\phi$, we cannot, in just topos logic, prove things like

- $\neg(\phi \wedge \psi) \Rightarrow (\neg\phi \vee \neg\psi)$
- $\neg\neg\phi \Rightarrow \phi$

nor any statements like

- $\neg(\forall x.\neg\phi) \Rightarrow (\exists x.\phi)$

DRAFT: Thursday 2nd February, 2012 16:49

- $\neg(\forall x.\phi) \Rightarrow (\exists x.\neg\phi)$
- $\neg(\exists x.\neg\phi) \Rightarrow (\forall x.\phi)$

We can, though, prove

- $\neg(\exists x.\phi) \Rightarrow (\forall x.\neg\phi)$

If we include, extra, an additional inference rule (called the *Boolean negation rule*) given by

$$\frac{\Gamma, \neg\phi : \perp}{\Gamma : \phi} \quad \frac{\Gamma : \phi}{\Gamma, \neg\phi : \perp}$$

then suddenly we're back in classical logic, and can prove $\neg\neg\phi \Rightarrow \phi$ and $\phi \vee \neg\phi$.

Examples: Sheaves, topology and time sheaves

The first interesting example of a topos is the category of (small enough) sets; in some sense clear already since we've been modelling our axioms and workflows pretty closely on the theory of sets.

Generating logic and set theory in the topos of sets, we get a theory that captures several properties of *intuitionistic logic*; such as the lack of Boolean negation, of exclusion of the third, and of double negation rules.

For the more interesting examples, however, we shall introduce the concepts of *topology* and of *sheaf*:

Definition A (set-valued) *presheaf* on a category C is a contravariant functor $E : C^{op} \rightarrow Set$.

Presheaves occur all over the place in geometry and topology - and occasionally in computer science too: There is a construction in which a functor $A \rightarrow Set$ for a discrete small category A identified with its underlying set of objects as a set, corresponds to the data type of *bags* of elements from A - for $a \in A$, the image $F(a)$ denotes the multiplicity of a in the bag.

Theorem The category of all presheaves (with natural transformations as the morphisms) on a category C form a topos.

Example Pick a category on the shape

$$0 \Longrightarrow 1$$

A contravariant functor on this category is given by a pair of sets G_0, G_1 and a pair of function source, target : $G_1 \rightarrow G_0$. Identities are sent to identities.

The category of presheaves on this category, thus, is the category of graphs. Thus graphs form a topos.

The subobject classifier in the category of graphs is a graph with two nodes: in and out, and five arrows:

- $in \rightarrow^{all} in$
- $in \rightarrow^{both} in$
- $in \rightarrow^{source} out$
- $out \rightarrow^{target} in$
- $out \rightarrow^{neither} out$

Now, given a subgraph $H \leq G$, we define a function $\chi_H : G \rightarrow \Omega$ by sending nodes to in or out dependent on their membership. For an arrow a , we send it to all if the arrow is in H , and otherwise we send it to both/source/target/neither according to where its source and target reside.

To really get into sheaves, though, we introduce more structure - specifically, we define what we mean by a *topology*:

Definition Suppose P is a partially ordered set. We call P a *complete Heyting algebra* if

- There is a top element 1 such that $x \leq 1 \forall x \in P$.
- Any two elements x, y have an infimum (greatest lower bound) $x \wedge y$.
- Every subset $Q \subseteq P$ has a supremum (least upper bound) $\bigvee_{p \in P} p$.
- $x \wedge (\bigvee y_i) = \bigvee x \wedge y_i$

Note that for the partial order by inclusion of a family of subsets of a given set, being a complete Heyting algebra is the same as being a topology in the classical sense - you can take finite unions and any intersections of open sets and still get an open set.

If $\{x_i\}$ is a subset with supremum x , and E is a presheaf, we get functions $e_i : E(x) \rightarrow E(x_i)$ from functoriality. We can summarize all these e_i into $e = \prod_i e_i : E(x) \rightarrow \prod_i E(x_i)$.

Furthermore, functoriality gives us families of functions $c_{ij} : E(x_i) \rightarrow E(x_i \wedge x_j)$ and $d_{ij} : E(x_j) \rightarrow E(x_i \wedge x_j)$. These can be collected into $c : \prod_i E(x_i) \rightarrow \prod_{i,j} E(x_i \wedge x_j)$ and $d : \prod_j E(x_j) \rightarrow \prod_{i,j} E(x_i \wedge x_j)$.

Definition A presheaf E on a Heyting algebra is called a *sheaf* if $x = \bigvee x_i$ implies that

$$E(x) \xrightarrow{\prod_i c_i} \prod_i E(x_i) \xrightarrow{\prod_j d_j} \prod_{i,j} E(x_i \wedge x_j)$$

is an equalizer. If you have seen sheaves before, you may recognize this as the covering axiom.

In other words, E is a sheaf if whenever $x = \bigvee x_i$ and $c(\alpha) = d(\alpha)$, then there is some $\bar{\alpha}$ such that $\alpha = e(\bar{\alpha})$.

Theorem The category of sheaves on a Heyting algebra is a topos.

For context, we can think of sheaves over Heyting algebras as sets in a logic with an expanded notion of truth. Our Heyting algebra is the collection of truth values, and the sheaves are the fuzzy sets with fuzziness introduced by the Heyting algebra.

Recalling that subsets and predicates are viewed as the same thing, we can view the set $E(p)$ as the part of the fuzzy set E that is at least p true.

As it turns out, to really make sense of this approach, we realize that *equality* is a predicate as well - and thus can hold or not depending on the truth value we use.

Definition Let P be a complete Heyting algebra. A P -valued set is a pair (S, σ) of a set S and a function $\sigma : S \rightarrow P$. A category of fuzzy sets is a category of P -valued sets. A morphism $f : (S, \sigma) \rightarrow (T, \tau)$ of P -valued sets is a function $f : S \rightarrow T$ such that $\tau \circ f = \sigma$.

From these definitions emerges a fuzzy set theory where all components of it being a kind of set theory emerges from the topoidal approach above. Thus, say, subsets in a fuzzy sense are just monics, thus are injective on the set part, and such that the valuation, on the image of the injection, increases from the previous valuation: $(T, \tau) \subseteq (S, \sigma)$ if $T \subseteq S$ and $\sigma|_T = \tau$.

To get to topoi, though, there are a few matters we need to consider. First, we may well have several versions of the empty set - either a bona fide empty set, or just a set where every element is never actually there. This issue is minor. Much more significant though, is that while we can easily make (S, σ) give rise to a presheaf, by defining

$$E(x) = \{s \in S : \sigma(s) \geq x\}$$

this definition will not yield a sheaf. The reason for this boils down to $E(0) = S \neq 1$. We can fix this, though, by adjoining another element - \perp - to P giving P^+ . The new element \perp is imbued with two properties: it is smaller, in P^+ , than any other element, and it is mapped, by E to 1.

Theorem The construction above gives a fuzzy set (S, σ) the structure of a sheaf on the augmented Heyting algebra.

Corollary The category of fuzzy sets for a Heyting algebra P forms a topos.

Final note While this construction allows us to make *membership* a fuzzy concept, we're not really done fuzzy-izing sets. There are two fundamental predicates on sets: equality and membership. While fuzzy set theory, classically, only allows us to

make one of these fuzzy, topos theory allows us - rather easily - to make both these predicates fuzzy. Not only that, but membership reduces - with the power object construction - to equality testing, by which the fuzzy set theory ends up somewhat inconsistent in its treatment of the predicates.

Literature

At this point, I would warmly recommend the interested reader to pick up one, or more, of:

- Steve Awodey: Category Theory
- Michael Barr & Charles Wells: Categories for Computing Science
- Colin McLarty: Elementary Categories, Elementary Toposes

or for more chewy books

- Peter T. Johnstone: Sketches of an Elephant: a Topos Theory compendium
- Michael Barr & Charles Wells: Toposes, Triples and Theories

Exercises

No homework at this point. However, if you want something to think about, a few questions and exercises:

1. Prove the relations showing that \leq_1 is indeed a partial order on $[U \rightarrow \Omega]$.
2. Prove the universal quantifier theorem.
3. The *extension* of a formula ϕ over a list of variables x is the sub-object of the product of domains $A_1 \times \cdots \times A_n$ for the variables $x_1, \dots, x_n = x$ classified by the interpretation of ϕ as a morphism $A_1 \times \cdots \times A_n \rightarrow \Omega$. A formula is *true* if it classifies the entire product. A *sequent*, written $\Gamma : \phi$ is the statement that using the set of formulae Γ we may prove ϕ , or in other words that the intersection of the extensions of the formulae in Γ is contained in the extension of ϕ . If a sequent $\Gamma : \phi$ is true, we say that Γ *entails* ϕ . (some of the questions below are almost embarrassingly immediate from the definitions given above. I include them anyway, so that a *catalogue* of sorts of topoidal logic inferences is included here)

DRAFT: Thursday 2nd February, 2012 16:49

-
- a) Prove the following entailments:
- i. Trivial sequent: $\phi : \phi$
 - ii. True: $: true$ (note that true classifies the entire object)
 - iii. False: $false : \phi$ (note that false classifies the global minimum in the preorder of subobjects)
- b) Prove the following inference rules:
- i. Implication: $\Gamma, \phi : \psi$ is equivalent to $\Gamma : \phi \Rightarrow \psi$.
 - ii. Thinning: $\Gamma : \phi$ implies $\Gamma, \psi : \phi$
 - iii. Cut: $\Gamma, \psi : \phi$ and $\Gamma : \psi$ imply $\Gamma : \phi$ if every variable free in ψ is free in Γ or in ϕ .
 - iv. Negation: $\Gamma, \phi : false$ is equivalent (implications both ways) to $\Gamma : \neg\phi$.
 - v. Conjunction: $\Gamma : \phi$ and $\Gamma : \psi$ together are equivalent to $\Gamma : \phi \wedge \psi$.
 - vi. Disjunction: $\Gamma, \phi : \theta$ and $\Gamma, \psi : \theta$ together imply $\Gamma, \phi \vee \psi : \theta$.
 - vii. Universal: $\Gamma : \phi$ is equivalent to $\Gamma : \forall x. \phi$ if x is not free in Γ .
 - viii. Existential: $\Gamma, \phi : \psi$ is equivalent to $\Gamma, \exists x. \phi : \psi$ if x is not free in Γ or ψ .
 - ix. Equality: $: q = q$.
 - x. Biconditional: $(v \Rightarrow w) \wedge (w \Rightarrow v) : v = w$. We usually write $v \Leftrightarrow w$ for $v = w$ if $v, w : A \rightarrow \Omega$.
 - xi. Product: $p_1 u = p_1 u', p_2 u = p_2 u' : u = u'$ for $u, u' \in A \times B$.
 - xii. Product revisited: $: (p_1(s \times s') = s) \wedge (p_2(s \times s') = s')$.
 - xiii. Extensionality: $\forall x \in A. f(x) = g(x) : f = g$ for $f, g \in [A \rightarrow B]$.
 - xiv. Comprehension: $(\lambda x \in A. s)x = s$ for $x \in A$.
- c) Prove the following results from the above entailments and inferences -- or directly from the topoidal logic mindset:
- i. $: \neg(\phi \wedge \neg\phi)$.
 - ii. $: \phi \Rightarrow \neg\neg\phi$.
 - iii. $: \neg(\phi \vee \psi) \Rightarrow (\neg\phi \wedge \neg\psi)$.
 - iv. $: (\neg\phi \wedge \neg\psi) \Rightarrow \neg(\phi \wedge \psi)$.
 - v. $: (\neg\phi \vee \neg\psi) \Rightarrow \neg(\phi \vee \psi)$.
 - vi. $\phi \wedge (\theta \vee \psi)$ is equivalent to $(\phi \wedge \theta) \vee (\phi \wedge \psi)$.
 - vii. $\forall x. \neg\phi$ is equivalent to $\neg\exists x. \phi$.
 - viii. $\exists x \phi \Rightarrow \neg\forall x. \neg\phi$.

- ix. $\exists x \neg \phi \Rightarrow \neg \forall x. \phi$.
 - x. $\forall x \phi \Rightarrow \neg \exists x. \neg \phi$.
 - xi. $\phi : \psi$ implies $\neg \psi : \neg \phi$.
 - xii. $\phi : \psi \Rightarrow \phi$.
 - xiii. $\phi \Rightarrow \not\vdash : \not\vdash$.
 - xiv. $\not\vdash \vee \psi : \phi \Rightarrow \psi$ (but not the converse!).
 - xv. $\neg \neg \neg \phi$ is equivalent to $\neg \phi$.
 - xvi. $(\phi \wedge \psi) \Rightarrow \theta$ is equivalent to $\phi \Rightarrow (\psi \Rightarrow \theta)$ (currying!).
- d) Using the Boolean negation rule: $\Gamma, \neg \phi : \text{false}$ is equivalent to $\Gamma : \phi$, prove the following additional results:
- i. $\neg \neg \phi : \phi$.
 - ii. $: \phi \vee \neg \phi$.
- e) Show that either of the three rules above, together with the original negation rule, implies the Boolean negation rule.
- i. The converses of the three existential/universal/negation implications above.
- f) The restrictions introduced for the cut rule above block the deduction of an entailment: $\forall x. \phi \Rightarrow \exists x. \phi$. The issue at hand is that A might not actually have members; so choosing one is not a sound move. Show that this entailment can be deduced from the premise $\exists x \in A. x = x$.
- g) Show that if we extend our ruleset by the quantifier negation rule $\forall x \Leftrightarrow \neg \exists x. \neg$, then we can derive the entailment: $\forall w : w = t \vee w = \text{false}$. From this derive $: \phi \vee \neg \phi$ and hence conclude that this extension gets us Boolean logic again.
4. A *topology* on a topos E is an arrow $j : \Omega \rightarrow \Omega$ such that $j \circ \text{true} = \text{true}$, $j \circ j = j$ and $j \circ \wedge = \wedge \circ j \times j$. For a subobject $S \subseteq A$ with characteristic arrow $\chi_S : A \rightarrow \Omega$, we define its j -closure as the subobject $\bar{S} \subseteq A$ classified by $j \circ \chi_S$.
- a) Prove:
- i. $S \subseteq \bar{S}$.
 - ii. $\bar{\bar{S}} = \bar{S}$.
 - iii. $S \bar{\cap} T = \bar{S} \cap \bar{T}$.
 - iv. $S \subseteq T$ implies $\bar{S} \subseteq \bar{T}$.
 - v. $f^{-1}(S) = f^{-1}(\bar{S})$.

DRAFT: Thursday 2nd February, 2012 16:49

Please do not circulate

-
- b) We define S to be j -closed if $S = \bar{S}$. It is j -dense if $\bar{S} = A$. These terms are chosen due to correspondences to classical pointset topology for the topos of sheaves over some space. For a logical standpoint, it is more helpful to look at j as a modality operator: "*it is j -locally true that*" Given any $u : 1 \rightarrow \Omega$, prove that the following are topologies:
- i. $(u \rightarrow -) : \Omega \rightarrow \Omega$ (the *open topology*, where such a u in a sheaf topos ends up corresponding to an open subset of the underlying space, and the formulae picked out are true on at least all of that subset).
 - ii. $u \vee - : \Omega \rightarrow \Omega$ (the closed topology, where a formula is true if its disjunction with u is true -- corresponding to formulae holding over at least the closed set complementing the subset picked out)
 - iii. $\neg \neg : \Omega \rightarrow \Omega$. This may, depending on the topos, end up being interpreted as *true so far as global elements are concerned*, or *not false on any open set*, or other interpretations.
 - iv. 1_Ω .
- c) For a topos E with a topology j , we define an object A to be a *sheaf* iff for every X and every j -dense subobject $S \subseteq X$ and every $f : S \rightarrow A$ there is a unique $g : X \rightarrow A$ with $f = g \circ s$. In other words, A is an object that cannot see the difference between j -dense subobjects and objects. We write E_j for the full subcategory of j -sheaves.
- i. Prove that any object is a sheaf for 1_Ω .
 - ii. Prove that a subobject is dense for $\neg \neg$ iff its negation is empty. Show that $true + false : 1 + 1 \rightarrow \Omega$ is dense for this topology. Conclude that $1 + 1$ is dense in $\Omega_{\neg \neg}$ and thus that $E_{\neg \neg}$ is Boolean.