Introduction to categorical logic

Peter Selinger

Dalhousie University
Halifax, Canada

Categorical logic

Categorical logic is about the connections between the following three areas:

- Logic (more precisely, proof theory),
- Computation (more precisely, programming languages),
- Category theory.

Our starting point: computation.

Part I: Introductory examples

Describing behavior

Semantics: to give a mathematical description of the *behavior* of computer programs.

Method 1: (operational) Define a particular kind of machine (Turing machine, Von Neumann machine, Abstract machine, Virtual machine...). Then describe how to run each program on this machine.

Method 2: (denotational) Give a mathematical description of the behavior, independenly of any machine. Specifically, define some mathematical space of behaviors, then map each program to a point in that space.

What is a "mathematical description"?

Part of the basic fabric of **mathematics** (i.e., what every mathematician learns near the beginning of their education) is *how to encode various mathematical objects* (finite sets, integers, rational numbers, real numbers, cartesian coordinates, geometric objects, algebras, topologies, equivalence relations, etc.) *in set theory*. We learn the *standard encodings*, and we also learn how to create *new encodings*.

People often assume that **computer science** is about programming some machine, for example the Intel Core i5-3570 processor running the Windows 7 operating system.

But in fact, many parts of computer science can also be developed by *encoding various computing concepts* (functions, data types, computational effects) *in set theory*.

What is a behavior of a computer program?

Set-theoretic (functional) interpretation:

- A *type* is a *set*. Examples:
  - $\mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$.
  - $\mathbb{N} = \{0, 1, 2, \ldots\}$.
  - $\mathsf{String} = \{"", "a", "b", "ab", \ldots\}$.

- The behavior of a *program* with inputs A and outputs B is given by a *function*

$$f : A \to B.$$

Note: in this functional notion of behavior, some aspects of the program are lost, for example: How *long* does it take to compute $f(a)$? Two programs are considered equal if they compute equal outputs on equal inputs. This is called the *extensional* view of behavior.

Examples from different programming languages

- In **C** or **Java**:

```
int f(int x) {
  return x + 1;
}
```

- In **Haskell**:

```
f :: int -> int
f x = x+1
```

- In **Mathematica**:

```
f[x_] := x + 1
```

- In **lambda calculus**:

$f = \lambda x.x + 1$

All define the same function $f : \mathbb{N} \to \mathbb{N}$, namely $f(x) = x + 1$.

Compositionality

Programs are built up from smaller programs by means of *combinators*.

The principle of *compositionality* states that the behavior of the whole is uniquely determined by the behavior of the parts.

Therefore, parts that have equal behavior are *interchangeable*.

For example, the expressions $f(x) = (2x + 4)/2 - 2$ and $f(x) = x + 1$ are interchangeable.

For now, we only need to consider two combinators (more may be added later): *identity* and *composition*.

$$\mathsf{id} : A \to A \qquad\qquad \frac{f : A \to B \quad g : B \to C}{g \circ f : A \to C}$$

Computational effects

The idea of a program as a function is only a first approximation. In reality, programs do more than just mapping inputs to outputs. For example, they may:

- not terminate;
- be non-deterministic;
- make probabilistic choices;
- write to a file or read from a file;
- be interactive;
- read and modify global variables;
- raise an exception or generate an error;
- ...

Any such additional behaviors are called "*computational effects*".

Non-termination

Potentially non-terminating programs are easy to model. A program with input A and output B is now described as a *partial function* $f : A \rightharpoonup B$.

Concretely, let $\bot$ be a symbol that is not an element of any type. The behavior of a potentially non-terminating program is described as a function

$$f : A \to B + \bot$$

with the information interpretation $f(a) = b$ if f terminates on input a with output b, and $f(a) = \bot$ if f diverges.

Notations: $A + B$ denotes disjoint union of sets $A \dot{\cup} B$. We wrote $A + \bot$ instead of $A + \{\bot\}$.

Non-termination, continued

We also need to account for compositionality, i.e.: what happens to non-termination when programs are combined?

$$\mathsf{id}_\perp : A \to A + \perp \qquad\qquad \frac{f : A \to B + \perp \quad g : B \to C + \perp}{g \circ_\perp f : A \to C + \perp}$$

It is clear how to define the operations $\mathsf{id}_\perp$ and $\circ_\perp$:

- $\mathsf{id}_\perp(a) = a$ (the identity program always terminates)
- $(g \circ_\perp f)(a) = \begin{cases} g(b) & \text{if } f(a) = b, \\ \perp & \text{if } f(a) = \perp. \end{cases}$

  (a composition terminates iff each of the parts terminates)

Non-determinism

A program is *non-deterministic* if it may potentially return a different output each time it is run. For example, a program that computes the root of a polynomial might find a different root on different runs — or maybe it will always find the same root, but it is unspecified which one it finds.

Let $\mathscr{P}^+(A) = \{X \mid X \subseteq A, X \neq \emptyset\}$ denote the *non-empty powerset of A*.

We can describe the behavior of a non-deterministic program with input type A and output type B as a function

$$f : A \to \mathscr{P}^+(B)$$

with the informal interpretation: $f(a) = b_1, \ldots, b_n$ if f may non-deterministically return any of the outputs $b_1, \ldots, b_n$ on input a.

Non-determinism, continued

$\mathsf{id}_{nd} : A \to \mathscr{P}^+(A)$

$$\frac{f : A \to \mathscr{P}^+(B) \qquad g : B \to \mathscr{P}^+(C)}{g \circ_{nd} f : A \to \mathscr{P}^+(C)}$$

How do non-deterministic programs compose?

- $\mathsf{id}_{nd}(a) = \{a\}$ (the identity is deterministic)
- $(g \circ_{nd} f)(a) = \bigcup\{g(b) \mid b \in f(a)\}$.

  (apply g to every possible output of f)

Probabilistic computation

A program is *probabilistic* if is has access to a random number generator. For example, a probabilistic program might output true with probability $\frac{1}{3}$ and false with probabililty $\frac{2}{3}$.

Let $\mathsf{Pr}(A)$ denote the set of *probability distributions* on A.

We can describe the behavior of a probabilistic program with input type A and output type B as a function

$$f : A \to \mathsf{Pr}(B)$$

with the informal interpretation: $f(a)(b) = p$ if $f(a)$ returns b with probability p.

(Note: for simplicity, assume all probability distributions are countably supported.)

Probabilistic computation, continued

$$\mathrm{id}_{\mathrm{pr}} : A \to \mathrm{Pr}(A) \qquad\qquad \frac{f : A \to \mathrm{Pr}(B) \quad g : B \to \mathrm{Pr}(C)}{g \circ_{\mathrm{pr}} f : A \to \mathrm{Pr}(C)}$$

How do probabilistic programs compose?

- $\mathrm{id}_{\mathrm{pr}}(a)(b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$ (the identity is deterministic)
- $(g \circ_{\mathrm{pr}} f)(a)(c) = \sum_b f(a)(b) \cdot g(b)(c)$ (sum over all paths)

Output to a terminal

A computer program might write some characters while it runs; for example, to a terminal (console) or to a file.

Let Σ be the set of *characters* (for example, the ASCII alphabet; we will use $\Sigma = \{a, b, c\}$).

Let Σ^* denote the set of *strings*, i.e., finite sequences of elements of Σ. We will write ϵ for the empty string, and $s \cdot t$ for concatenation of strings. Example: $"ab" \cdot "bc" = "abbc"$.

We describe the behavior of a program with input type A, output type B, and writing some characters to a terminal, as a function

$$f : A \to B \times \Sigma^*,$$

with the informal interpretation: $f(a) = (b, s)$ if $f(a)$ writes s and returns b.

Output to a terminal, continued

$$\text{id}_{\text{out}} : A \to A \times \Sigma^*$$

$$\frac{f : A \to B \times \Sigma^* \quad g : B \to C \times \Sigma^*}{g \circ_{\text{out}} f : A \to C \times \Sigma^*}$$

How do such programs compose?

- $\text{id}_{\text{out}}(a) = (a, \epsilon)$ (the identity function writes nothing)
- $(g \circ_{\text{out}} f)(a) = (c, s \cdot t)$ where $f(a) = (b, s)$ and $g(b) = (c, t)$

  (f writes first, g writes second)

State

A program is *stateful* if it has access to some global *state* (for example, some global variables) that it may read and update. For example, a program may increment a counter, and use this to return a different integer each time it is called.

Let S be the set of states.

We can describe the behavior of a stateful program with input type A and output type B as a function

$$f : A \times S \to B \times S$$

with the informal interpretation: $f(a, s_1) = (b, s_2)$ if the program f with input a, run in state s_1, produces output b and updates the state to s_2.

State, continued

$\mathsf{id}_{st} : A \times S \to A \times S$

$$\frac{f : A \times S \to B \times S \quad g : B \times S \to C \times S}{g \circ_{st} f : A \times S \to C \times S}$$

How do stateful programs compose?

- $\mathsf{id}_{st}(a, s) = (a, s)$ (the identity does not update the state)
- $(g \circ_{st} f)(a, s_1) = (c, s_3)$
  where $f(a, s_1) = (b, s_2)$ and $g(b, s_2) = (c, s_3)$.

  (first f updates the state, then g is run in this new state)

Computational effects and monads

What do all these examples have in common? Eugenio Moggi observed that computational effects all have the structure of a *monad*.

In each case, we have some operation T on sets:

- $\mathsf{T}(A) = A + \bot$ (non-termination)
- $\mathsf{T}(A) = \mathscr{P}^{+}(A)$ (non-determinism)
- $\mathsf{T}(A) = \mathsf{Pr}(A)$ (probabilistic)
- $\mathsf{T}(A) = A \times \Sigma^{*}$ (terminal output)
- ...

Computational effects and monads, continued

In each case, we define a *function with computational effects*, with input type A and output type B, to be a set-theoretic function

$$f : A \to T(B).$$

Finally, in each case, we define an effectful identity and an effectful composition:

$$\mathsf{id}_T : A \to T(A) \qquad\qquad \frac{f : A \to T(B) \quad g : B \to T(C)}{g \circ_T f : A \to T(C)}$$

For this to make any sense, the operations T, id_T, and $\circ_T$ must satisfy certain properties, for example

$$\mathsf{id}_T \circ_T f = f, \quad g \circ_T \mathsf{id}_T = g, \quad h \circ_T (g \circ_T f) = (h \circ_T g) \circ_T f.$$

Such a structure $(T, \mathsf{id}_T, \circ_T)$ is called a *monad*.

The state monad

One of our examples does not seem to fit the pattern of a monad. Namely, in the case of stateful computation, we used:

$$f : A \times S \to B \times S.$$

However, this can easily be rewritten to fit the same pattern as the other examples:

$$f : A \to (B \times S)^S.$$

Here, $X^Y = \{g \mid g : Y \to X\}$ denotes the set of all functions from Y to X.

We therefore have the *state monad*

$$T(A) = (A \times S)^S.$$

Part II: Introduction to category theory

Categories

A *category* $\mathbf{C}$ consists of:

- A collection $|\mathbf{C}|$ of *objects* A, B, C, ...

- For each pair A, B of objects, a set of *morphisms*

$$\mathbf{C}(A, B)$$

  We also write $f : A \rightarrow B$ to indicate $f \in \mathbf{C}(A, B)$.

- with *operations*

$$\frac{f : A \rightarrow B \qquad g : B \rightarrow C}{g \circ f : A \rightarrow C} \qquad \frac{}{\mathsf{id}_A : A \rightarrow A}$$

  Note: this notation just means:

$$\circ : \mathbf{C}(B, C) \times \mathbf{C}(A, B) \rightarrow \mathbf{C}(A, C),$$
$$\mathsf{id}_A \in \mathbf{C}(A, A).$$

Categories, continued

. . .

- subject to the *equations*:

$$\mathsf{id}_B \circ f = f, \qquad f \circ \mathsf{id}_A = f, \qquad (h \circ g) \circ f = h \circ (g \circ f).$$

Examples of categories

- the category **Set** of *sets* (and functions),
- the category **Rel** of *sets* (and relations),
- the category **Grp** of *groups* (and homomorphisms),
- the category **Ab** of *abelian groups* (and homomorphisms),
- the category **Rng** of *rings* (and ring homomorphisms),
- the category **Vec** of *vector spaces* (and linear functions),
- the category **Top** of *topological spaces* (and continuous functions),
- *logic:* objects = propositions, morphisms = proofs
- *computing:* objects = data types, morphisms = programs

Concepts such as *inverse*, *monomorphism* (injection), *idempotent*, *product*, etc, make sense in any category.

Functors

Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A *functor* $F : \mathbf{C} \to \mathbf{D}$ is given by the following data:

- A function $F : |\mathbf{C}| \to |\mathbf{D}|$ from the objects of $\mathbf{C}$ to the objects of $\mathbf{D}$;
- For every pair of objects $A, B \in |\mathbf{C}|$, a function $F : \mathbf{C}(A, B) \to \mathbf{D}(FA, FB)$;
- subject to the equations

$$F(\mathsf{id}_A) = \mathsf{id}_{FA}, \quad F(g \circ f) = Fg \circ Ff$$

Note: we use F to denote both the object part and the morphism part of the functor. We also often write FA, Ff, etc., instead of the more traditional $F(A)$, $F(f)$.

Examples of functors

On **Set**:

- $F(A) = A + X$ (where X is a fixed set)
- $F(A) = \mathscr{P}(A)$ (powerset)
- $F(A) = \mathscr{P}^{+}(A)$ (non-empty powerset)
- $F(A) = \mathsf{Pr}(A)$ (probability)
- $F(A) = A \times X$ (where X is a fixed set)
- $F(A) = A \times A$
- $F(A) = A^{X}$ (where X is a fixed set)
- $F(A) = X$ (where X is a fixed set: constant functor)
- $F(A) = A^{*}$ (list functor)

Exercise: supply the missing data making each of these examples into a functor. A priori this is not unique!

Examples of functors from mathematics

- $F : \mathbf{Grp} \to \mathbf{Set}$ given by $F(G) = |G|$, the "underlying set" of the group, and $F(\phi) = \phi$. This is called a "forgetful" functor.

- There are also forgetful functors $\mathbf{Rng} \to \mathbf{Grp}$, $\mathbf{Ring} \to \mathbf{Ab}$, $\mathbf{Ab} \to \mathbf{Grp}$, $\mathbf{Top} \to \mathbf{Set}$, and so on.

- $F : \mathbf{Set} \to \mathbf{Grp}$, where $F(X)$ is the *free group* generated by X.

- $F : \mathbf{Set} \to \mathbf{Vec}$, where $F(X)$ is the vector space with basis X.

- $F : \mathbf{Top}_* \to \mathbf{Grp}$, where $F(X) = \pi_1(X)$ is the *fundamental group* of X.

Exercise: supply the missing data, and check that each of these is a functor.

Natural transformations

Let $\mathbf{C}, \mathbf{D}$ be categories and let $F, G : \mathbf{C} \to \mathbf{D}$ be two functors. A *natural transformation* $\eta : F \to G$ is given by the following data:

- for every object $A \in |\mathbf{C}|$, a morphism $\eta_A : FA \to GA$;
- subject, for every $f : A \to B$ in $\mathbf{C}$, to the equation

$$\begin{array}{ccc} FA & \xrightarrow{Ff} & FB \\ {\scriptstyle \eta_A}\downarrow & & \downarrow{\scriptstyle \eta_B} \\ GA & \xrightarrow{Gf} & GB. \end{array}$$

Note: the diagram is just a notation for an equation

$$\eta_B \circ Ff = Gf \circ \eta_A.$$

Examples of natural transformations

On **Set**, let F be the list functor $F(A) = A^*$, and let G be the powerset functor $G(A) = \mathscr{P}(A)$.

The function $\eta_A : A^* \to \mathscr{P}(A)$ defined by

$$\eta_A(x_1, \ldots, x_n) = \{x_1, \ldots, x_n\}$$

is a natural transformation.

The function $\eta_A : A^* \to A^*$ defined by

$$\eta_A(x_1, \ldots, x_n) = (x_n, \ldots, x_1)$$

is a natural transformation.

The function $\eta_A : \mathscr{P}^{\mathrm{fin}}(A) \to A^*$ defined by

$$\eta_A\{x_1, \ldots, x_n\} = (x_1, \ldots, x_n)$$

(in some arbitrary but fixed order) is not a natural transformation.

Monads

Let $\mathbf{C}$ be a category. A *monad* (T, η, μ) on $\mathbf{C}$ is given by the following data:

- A functor $T : \mathbf{C} \to \mathbf{C}$;
- Two natural transformations $\eta : 1 \to T$ and $\mu : T^2 \to T$;
- subject to the equations

$$\begin{array}{ccc} T & \xrightarrow{\eta T} & T^2 \\ {\scriptstyle T\eta}\downarrow & \searrow{\scriptstyle \mathrm{id}} & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow{\mu} & T, \end{array} \qquad \begin{array}{ccc} T^3 & \xrightarrow{\mu T} & T^2 \\ {\scriptstyle T\mu}\downarrow & & \downarrow{\scriptstyle \mu} \\ T^2 & \xrightarrow{\mu} & T. \end{array}$$

The Kleisli category of a monad

Recall our compositionality requirement from Part I:

$$\mathrm{id}_T : A \to TA \qquad\qquad \frac{f : A \to TB \quad g : B \to TC}{g \circ_T f : A \to TC}$$

Given a monad (T, η, μ) on a category $\mathbf{C}$, we actually have enough data to define these operations. Specifically, we can define

- $\mathrm{id}_T = A \xrightarrow{\eta_A} TA$;
- $g \circ_T f = A \xrightarrow{f} TB \xrightarrow{Tg} T(TC) \xrightarrow{\mu_C} TC$.

Exercise: verify the three laws

$$\mathrm{id}_T \circ_T f = f, \quad g \circ_T \mathrm{id}_T = g, \quad h \circ_T (g \circ_T f) = (h \circ_T g) \circ_T f.$$

Exercise: show that these three laws are *equivalent* to the equations in the definition of a monad.

Kleisli category, continued

Let (T, η, μ) be a monad on a category $\mathbf{C}$. Recall that an "effectful" map from A to B is given by

$$f : A \to TB,$$

with identities and composition as on the previous slide. It is then natural to make a new category, with the same objects as $\mathbf{C}$, but using the "effectful" maps as the morphisms. This is called the *Kleisli category* of T, and denoted $\mathbf{C}_T$.

- Objects: $\mathbf{C}_T$ has the same objects as $\mathbf{C}$.
- Morphisms: $\mathbf{C}_T(A, B) = \mathbf{C}(A, TB)$.
- Identities and composition: as on the previous slide.

Composing functors

Horizontal composition (functors):

$$\mathbf{C} \xrightarrow{F} \mathbf{D} \xrightarrow{G} \mathbf{E}$$

If F, G are functors, then so is $G \circ F$. Defined on objects as $(G \circ F)(A) = G(F(A))$ and on morphisms as $(G \circ F)(f) = G(F(f))$.

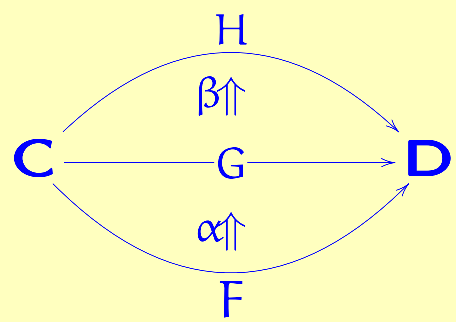
Identity (functors):

$$\mathbf{C} \xrightarrow{1_{\mathbf{C}}} \mathbf{C}$$

The identity functor $1_{\mathbf{C}} : \mathbf{C} \to \mathbf{C}$ is defined as $1_{\mathbf{C}}(A) = A$ on objects and $1_{\mathbf{C}}(f) = f$ on morphisms.

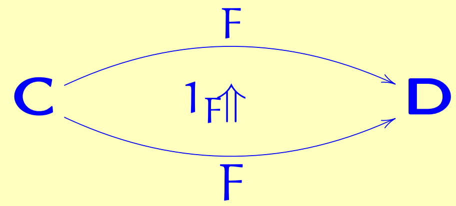
Composing natural transformations

Vertical composition (natural transformations):



If $\alpha : F \to G$ and $\beta : G \to H$ are natural transformations, then so is $\beta \bullet \alpha : F \to H$. If it defined by $(\beta \bullet \alpha)_A = \beta_A \circ \alpha_A : FA \to HA$.

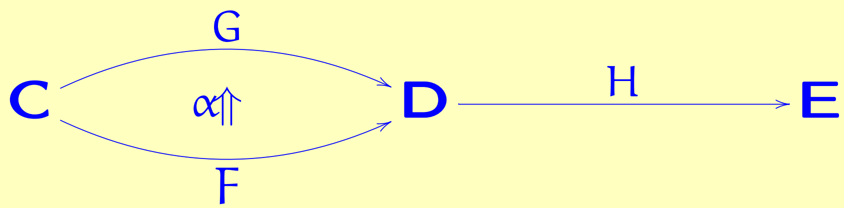
Identity (natural transformations):



The identity natural transf. $1_F : F \to F$ is defined as $(1_F)_A = 1_{FA}$. By abuse of notation, we sometimes denote 1_F by 1, or even F.

Composing natural transformations, continued

Whiskering (right):



If $F, G : \mathbf{C} \to \mathbf{D}$ and $H : \mathbf{D} \to \mathbf{E}$ are functors, and if $\alpha : F \to G$ is a natural transformation, the *right whiskering*

$$H \circ \alpha : H \circ F \to H \circ G$$

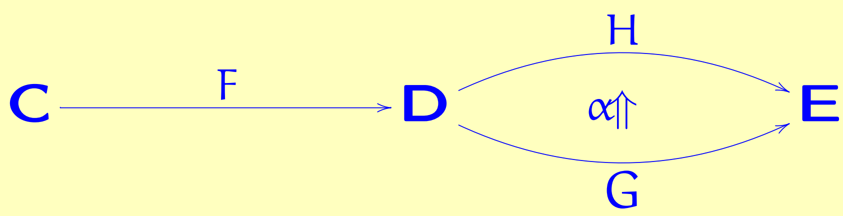
is defined as $(H \circ \alpha)_A : H(FA) \to H(GA)$ by $(H \circ \alpha)_A = H(\alpha_A)$. This is indeed a natural transformation, i.e.,

$$\begin{array}{ccc} H(FA) & \xrightarrow{H(\alpha_A)} & H(GA) \\ {\scriptstyle H(Ff)}\downarrow & & \downarrow{\scriptstyle H(Gf)} \\ H(FB) & \xrightarrow{H(\alpha_B)} & H(GB). \end{array}$$

In this case, it follows from the naturality of α and the functoriality of H.

Composing natural transformations, continued

Whiskering (left):



If $F : \mathbf{C} \to \mathbf{D}$ and $G, H : \mathbf{D} \to \mathbf{E}$ are functors, and if $\alpha : G \to H$ is a natural transformation, the *left whiskering*

$$\alpha \circ F : G \circ F \to H \circ F$$

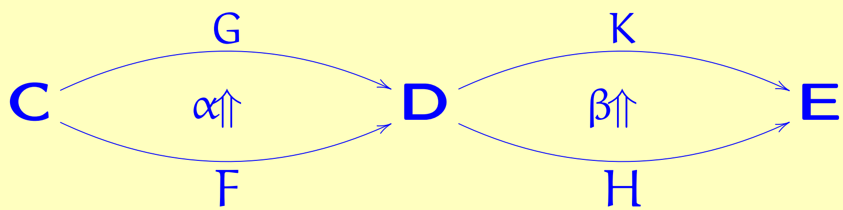
is defined as $(\alpha \circ F)_A : G(FA) \to H(FA)$ by $(\alpha \circ F)_A = \alpha_{FA}$. This is indeed a natural transformation, i.e.,

$$\begin{array}{ccc} G(FA) & \xrightarrow{\alpha_{FA}} & H(FA) \\ {\scriptstyle G(Ff)}\downarrow & & \downarrow{\scriptstyle H(Ff)} \\ G(FB) & \xrightarrow{\alpha_{FB}} & H(FB). \end{array}$$

In this case, it follows from the naturality of α.

Composing natural transformations, continued

Horizontal composition (natural transformations):



If $F, G : \mathbf{C} \to \mathbf{D}$ and $H, K : \mathbf{D} \to \mathbf{E}$ are functors, and if $\alpha : F \to G$ and $\beta : H \to K$ are natural transformations, the *horizontal composition*

$$\beta \circ \alpha : H \circ F \to K \circ G$$

can be defined in two different ways:

- Right whiskering followed by left whiskering: $\beta \circ \alpha = (\beta \circ G) \bullet (H \circ \alpha)$
- Left whiskering followed by right whiskering: $\beta \circ \alpha = (K \circ \alpha) \bullet (\beta \circ F)$.

Composing natural transformations, continued

- Right whiskering followed by left whiskering:
  $\beta \circ \alpha = (\beta \circ G) \bullet (H \circ \alpha)$

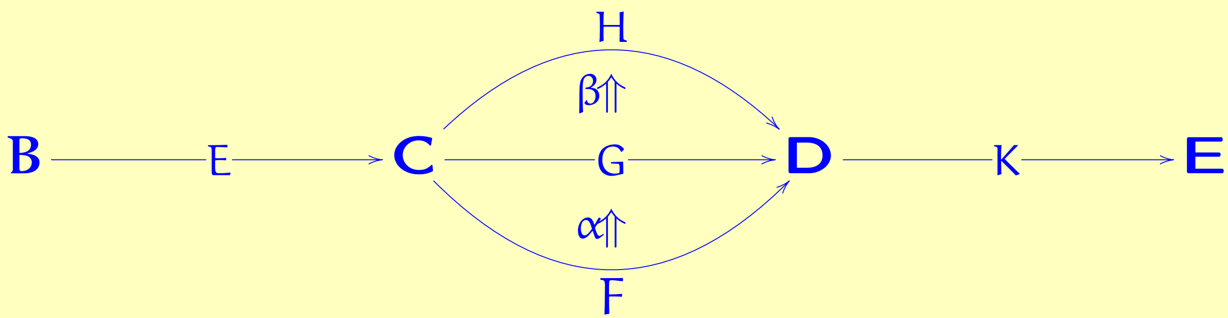
- Left whiskering followed by right whiskering:
  $\beta \circ \alpha = (K \circ \alpha) \bullet (\beta \circ F)$.

The two definitions coincide, because
$[(\beta \circ G) \bullet (H \circ \alpha)]_A = \beta_{GA} \circ H(\alpha_A)$, and
$[(K \circ \alpha) \bullet (\beta \circ F)]_A = K(\alpha_A) \circ \beta_{FA}$, and

$$\begin{array}{ccc} H(FA) & \xrightarrow{H(\alpha_A)} & H(GA) \\ {\scriptstyle \beta_{FA}} \downarrow & & \downarrow {\scriptstyle \beta_{GA}} \\ K(FA) & \xrightarrow[K(\alpha_A)]{} & K(GA). \end{array}$$

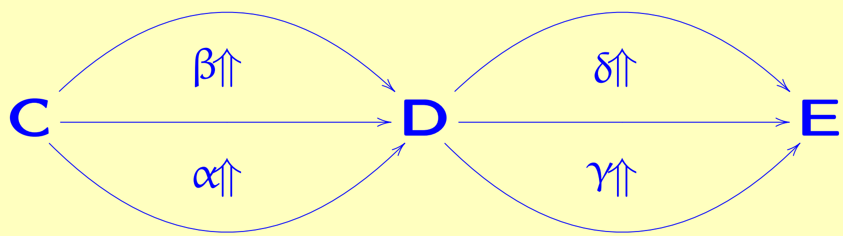
by naturality of β.

Some laws about whiskering



$$K \circ (\beta \bullet \alpha) = (K \circ \beta) \bullet (K \circ \alpha)$$
$$K \circ 1_F = 1_{K \circ F}$$
$$1_K \circ \alpha = K \circ \alpha$$
$$1_K \circ 1_F = 1_{K \circ F}$$
$$(\beta \bullet \alpha) \circ E = (\beta \circ E) \bullet (\alpha \circ E)$$
$$1_F \circ E = 1_{F \circ E}$$
$$\alpha \circ 1_E = \alpha \circ E$$

The double interchange law



$$(\delta \circ \beta) \bullet (\gamma \circ \alpha) = (\delta \bullet \gamma) \circ (\beta \bullet \alpha)$$

Example: The list monad

Recall that $F(A) = A^*$ is the *list monad*. Here A^* is the set of finite *lists* (also known as *words*, *strings*) of elements from A.

F is a monad as follows:

- Functor: for $f : A \to B$, define $f^* : A^* \to B^*$ by

$$f^*[a_1, \ldots, a_n] = [f(a_1), \ldots, f(a_n)].$$

- Unit: we define $\eta_A : A \to A^*$ by

$$\eta_A(a) = [a] \quad \text{(singleton)}.$$

- Multiplication: we define $\mu_A : A^{**} \to A^*$ by

$$\mu_A([l_1, l_2 \ldots, l_n]) = l_1 \cdot l_2 \cdot \ldots \cdot l_n.$$

Verify the monad laws.

Free algebras

Let Σ be a *signature*, and let E be a set of *equations* (both in the sense of universal algebra).

A signature consists of a set $|\Sigma| = \{f, g, \ldots\}$ of *function symbols*, together with an assignment $\mathsf{ar} : |\Sigma| \to \mathbb{N}$ of an *arity* to each function symbol.

Fix a signature. For example, let h be a function symbol of arity 2, and let g be a function symbol of arity 1.

Let V be a set of *variables*. Then we can form the set of *terms*, e.g.:

$$x, \quad y, \quad g(x), \quad g(y), \quad h(x, x), \quad h(x, y),$$
$$h(g(x), y), \quad h(g(g(x)), x), \quad g(h(x, g(h(y, x)))), \ldots$$

Let $\mathsf{Terms}_\Sigma(V)$ be this set of terms.

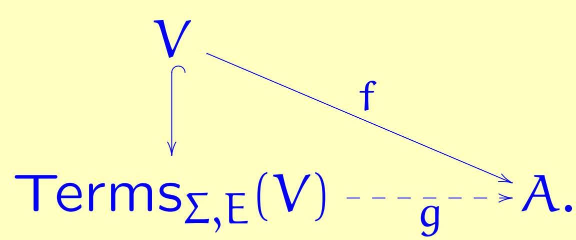
Free algebras, continued

On the set $\text{Terms}_\Sigma(V)$, consider the smallest equivalence relation $\sim_E$ such that:

$$\frac{(t = s) \in E}{t' \sim_E s'} \qquad \frac{t_1 \sim_E s_1, \quad \ldots, \quad t_n \sim_E s_n}{f(t_1, \ldots, t_n) \sim_E f(s_1, \ldots, s_n)}$$

Then $\text{Terms}_\Sigma(V)/\sim_E$ is a (Σ, E)-algebra. We denote it by $\text{Terms}_{\Sigma,E}(V)$.

In fact, it is the *free* (Σ, E)-algebra generated by V. Concretely, this means: for any (Σ, E)-algebra A, and any function $f : V \to A$, there exists a unique homomorphism of (Σ, E)-algebras $g : \text{Terms}_{\Sigma,E}(V) \to A$ such that



The term monad

Fix Σ and E. Consider the functor $T : \mathbf{Set} \to \mathbf{Set}$ given by

$$T(V) = \mathsf{Terms}_{\Sigma,E}(V).$$

This is a monad:

- Functor: for $f : V \to W$, define $T(f) : \mathsf{Terms}_{\Sigma,E}(V) \to \mathsf{Terms}_{\Sigma,E}(W)$ by "renaming" all the variables in a term.

- Unit: $\eta_V : V \to \mathsf{Terms}_{\Sigma,E}(V)$ maps a variable x to the term x.

- Multiplication: $\mu_V : T(T(V)) \to T(V)$ takes a term whose "variables" are other terms. It is defined by "flattening" this structure into a single term.

Check the monad laws!

The list monad as a term monad

In fact, the list monad $A \mapsto A^*$ is the term monad for operations "$\cdot$" (arity 2), e (arity 0), with equations

$$(x \cdot y) \cdot z = x \cdot (y \cdot z), \quad e \cdot x = x, \quad x \cdot e = x.$$

In other words, A^* is the *free monoid* on A. Also:

- $T(A) = A + \bot$ is the term monad over the signature $\Sigma = \{\bot\}$ (arity 0, no equations);
- $T(A) = \mathscr{P}^{\mathrm{fin}}(A)$ is like the list monad, with the additional equations

$$x \cdot x = x, \quad x \cdot y = y \cdot x;$$

- $T(A) = \mathscr{P}^{\mathrm{fin},+}$ is the same, but without the constant e;
- $T(A) = A \times \Sigma^*$ is the term monad over the signature $\{w_c \mid c \in \Sigma\}$, each with arity 1.

An alternative definition of monad [Manes]

Let **C** be a category, and let $T : |\mathbf{C}| \to |\mathbf{C}|$ be a function on objects (here *not* a priori assumed to be a functor).

Suppose that T is equipped with the following two operations:

$$\frac{}{\eta_A : A \to TA} \qquad \frac{f : A \to TB}{\mathrm{lift}(f) : TA \to TB}$$

Satisfying:

$(a)\ \mathrm{lift}(\eta_A) = 1_{TA} \quad (b)\ (\mathrm{lift} f) \circ \eta_A = f \quad (c)\ \mathrm{lift}((\mathrm{lift} g) \circ f) = (\mathrm{lift} g) \circ (\mathrm{lift} f)$

Note: then T can be made into a functor like this:

$$\frac{\dfrac{f : A \to B}{\eta_B \circ f : A \to TB}}{\mathrm{lift}(\eta_B \circ f) : TA \to TB}$$

Exercise: prove that this is an equivalent definition of monad.

Kleisli category of a monad: $\mathbf{C}_T$

Let (T, η, μ) be a monad on a category $\mathbf{C}$. Its *Kleisli category* $\mathbf{C}_T$ is defined as follows:

- Objects: $\mathbf{C}_T$ has the same objects as $\mathbf{C}$.
- Morphisms: $\mathbf{C}_T(A, B) = \mathbf{C}(A, TB)$.
- Identities and composition:

$$\mathsf{id}_T : A \to TA \qquad\qquad \frac{f : A \to TB \quad g : B \to TC}{g \circ_T f : A \to TC}$$

Then $\mathbf{C}_T$ is a well-defined category. Moreover, there is a canonical functor $F : \mathbf{C} \to \mathbf{C}_T$ mapping A to A and f to $\eta_B \circ f$, and a canonical functor $G : \mathbf{C}_T \to \mathbf{C}$ mapping A to TA and g to $\mathsf{lift}(g)$.

Algebras of a monad: $\mathbf{C}^T$

Let (T, η, μ) be a monad on a category $\mathbf{C}$.

Definition. An *algebra* for T is a pair (A, a), where A is an object of $\mathbf{C}$, and $a : TA \to A$ is a morphism, satisfing

$$\begin{array}{ccc} T^2A & \xrightarrow{Ta} & TA \\ {\scriptstyle \mu_A}\downarrow & & \downarrow{\scriptstyle a} \\ TA & \xrightarrow{a} & A, \end{array} \qquad \begin{array}{ccc} A & \xrightarrow{\eta_A} & TA \\ & {\scriptstyle 1_A}\searrow & \downarrow{\scriptstyle a} \\ & & A. \end{array}$$

Given two algebras (A, a) and (B, b), a *homomorphism* is given by a map $f : A \to B$ satisfying

$$\begin{array}{ccc} TA & \xrightarrow{Tf} & TB \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \xrightarrow{f} & B. \end{array}$$

Consider what this means in case of the term monad for (Σ, E).

Eilenberg-Moore category of a monad: $\mathbf{C}^T$

Let (T, η, μ) be a monad on a category $\mathbf{C}$. Its *Eilenberg-Moore category* $\mathbf{C}^T$ is defined as follows:

- Objects: algebras (A, a) for the monad T.
- Morphisms: algebra homomorphisms.
- Identities and composition: as in $\mathbf{C}$.

Then $\mathbf{C}^T$ is a well-defined category. Moreover, there is a canonical functor $F : \mathbf{C} \to \mathbf{C}^T$ mapping A to (TA, μ_A) and f to Tf. There is also a canonical functor $G : \mathbf{C}^T \to \mathbf{C}$ mapping (A, a) to A.

Some small categories

- Let $(P, \leq)$ be a *partially ordered set* (i.e., $\leq$ is reflexive, transitive, and antisymmetric). Then P is a category, where the *objects* are the elements of P, and there exists a *unique* morphism $f : x \to y$ iff $x \leq y$.

- Let $(M, \bullet, e)$ be a *monoid* (i.e., $\bullet$ is an associative operation with unit e). Then M is a category, where there is a *unique* object $*$, and one morphism $x : * \to *$ for each element $x \in M$, with composition $x \circ y = x \bullet y$ and identity $\mathrm{id} = e$.

- Let X be a *set*. Then X is a category, called the *discrete* category, where the objects are the elements of X, and the only morphisms are identities $\mathrm{id}_x : x \to x$.

Cartesian product of two categories

If $\mathbf{C}, \mathbf{D}$ are categories, then $\mathbf{C} \times \mathbf{D}$ is a category defined as:

- Objects: (A, B) where $A \in |\mathbf{C}|$ and $B \in |\mathbf{D}|$;
- Morphisms: $(f, g) : (A, B) \to (A', B')$ where $f : A \to A'$ and $g : B \to B'$;
- Composition and identities: componentwise.

Duality

If $\mathbf{C}$ is a category, then its *dual category* $\mathbf{C}^{\mathsf{op}}$ is defined by

- Objects: $\mathbf{C}^{\mathsf{op}}$ has the same objects as $\mathbf{C}$;
- Morphisms: $\mathbf{C}^{\mathsf{op}}(A, B) = \mathbf{C}(B, A)$;
- Identities: same as those of $\mathbf{C}$;
- Composition: in reverse order, i.e.: $g \circ_{\mathbf{C}^{\mathsf{op}}} f = f \circ_{\mathbf{C}} g$.

For every definition or theorem about categories, there is a *dual* definition or theorem, obtained by replacing the category by its dual.

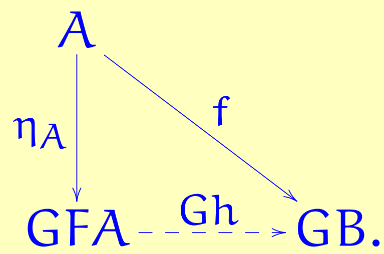
Adjunctions

Suppose $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{D} \to \mathbf{C}$ are two functors, and further assume that there is a *natural isomorphism of hom-sets*

$$\mathbf{D}(FA, B) \cong \mathbf{C}(A, GB).$$

Then F is called a *left adjoint* of G, and G is called a *right adjoint* of F. We write $F \dashv G$.

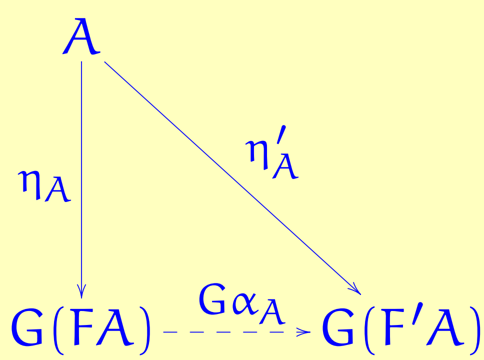
Equivalently (and more concretely), this means that there is $\eta_A : A \to G(FA)$, and for every $f : A \to GB$, there exists a unique $h : FA \to B$ satisfying



Adjoints arise everywhere in mathematics. For example: if G is a *forgetful functor*, and F is its left adjoint, then F is a *free functor*.

Uniqueness of adjoints

Theorem. Suppose that $G : \mathbf{D} \to \mathbf{C}$ is a functor, and that $F, F' : \mathbf{C} \to \mathbf{D}$ are two left adjoints of G. Then there exists a natural isomorphism $\alpha : F \to F'$ such that $\eta' = G\alpha \bullet \eta$.



Adjoints between posets

Remark. Let P, Q be partially ordered (or preordered) sets, and let $f : P \to Q$ and $g : Q \to P$ be monotone functions. Then f is left adjoint to g if and only if for all $x \in P$, $y \in Q$:

$$f(x) \leq y \quad \iff \quad x \leq g(y).$$

(Equivalently, f is "residuated", or f and g form a "Galois connection").

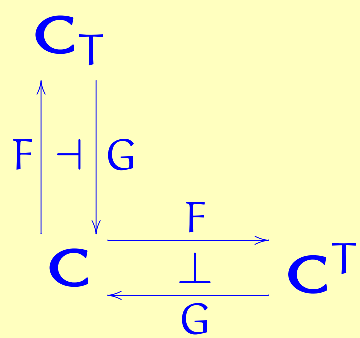
Adjunctions and monads

Every adjunction $F \dashv G$, where $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{D} \to \mathbf{C}$, defines a monad on $\mathbf{C}$, via

$$T = G \circ F.$$

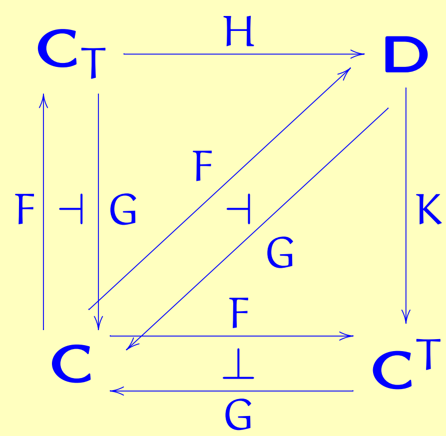
(Note: lots of details omitted).

Conversely, every monad arises in this way: actually in *two* different ways that are canonical: If T is any monad on a category $\mathbf{C}$, then both the *Kleisli construction* and the *Eilenberg-Moore construction* give rise to adjuctions, each satisfying $T = G \circ F$:



Adjunctions and monads, continued

Both the *Kleisli construction* and the *Eilenberg-Moore construction* give rise to adjuctions, each satisfying $T = G \circ F$. Moreover, they are *universal*: given any third adjunction $F \dashv G$ between $\mathbf{C}$ and some category $\mathbf{D}$, there exist *unique* functors $H : \mathbf{C}_T \to \mathbf{D}$ and $K : \mathbf{D} \to \mathbf{C}^T$ such that



Constructions <u>within</u> categories (blackboard)

- Monomorphism (dual: epimorphism)
- Isomorphism
- Terminal object (dual: initial object)
- Finite products (dual: coproducts)
- Equalizers (dual: coequalizers)
- Limits (dual: colimits)

Monomorphisms and epimorphisms

Let $f : A \to B$ be a morphism in a category. Then f is called a *monomorphism* (or *monic*) if:

for all objects X, and all morphisms $g, h : X \to A$,

$$f \circ g = f \circ h \quad \Rightarrow \quad g = h.$$

$$X \underset{h}{\overset{g}{\rightrightarrows}} A \xrightarrow{f} B.$$

The dual concept is called an *epimorphism* (or *epic*).

Isomorphisms

A morphism $f : A \to B$ in a category is called an *isomorphism* if it is invertible, i.e., there exists some $g : B \to A$ such that $f \circ g = 1_B$ and $g \circ f = 1_A$.

A natural transformation $\alpha : F \to G$ is called a *natural isomorphism* if $\alpha_A : FA \to GA$ is an isomorphism for all A.

A category in which *all* morphisms are invertible is called a *groupoid*. (Or in case there is only one object, it is called a *group*).

Example: in **Set**, monomorphism = injective, epimorphism = surjective, isomorphism = bijective.

In **Top**, monomorphism = injective, epimorphism = dense, isomorphism = homeomorphism.

Terminal object

An object A in a category is called *terminal* if:

for all objects X, there exists a *unique* morphism $g : X \to A$.

Note: a terminal object, if it exists, is unique up to isomorphism.

The dual concept is called an *initial* object.

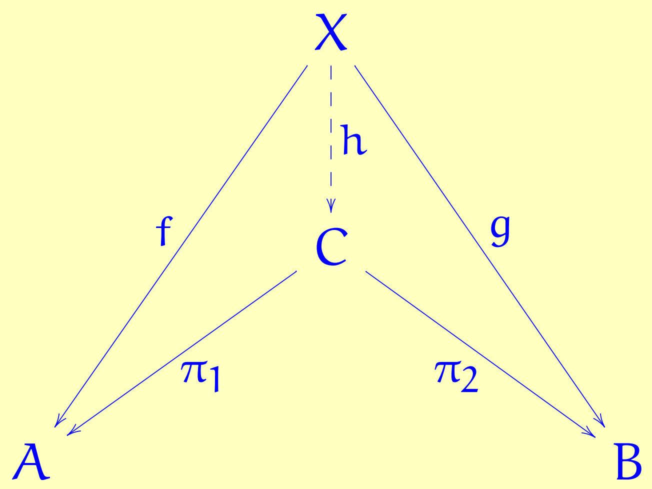
Example: in **Set**, $1 = \{*\}$ is terminal and $0 = \emptyset$ is initial.

In **Vec**, **Grp**, and $\mathbf{Set}_\perp$, 1 is initial and terminal.

Categorical product

Let A, B be objects in a category. A *categorical product* of A and B is a triple (C, π_1, π_2), where C is an object, $\pi_1 : C \to A$ and $\pi_2 : C \to B$ are morphisms, and such that the following property holds:

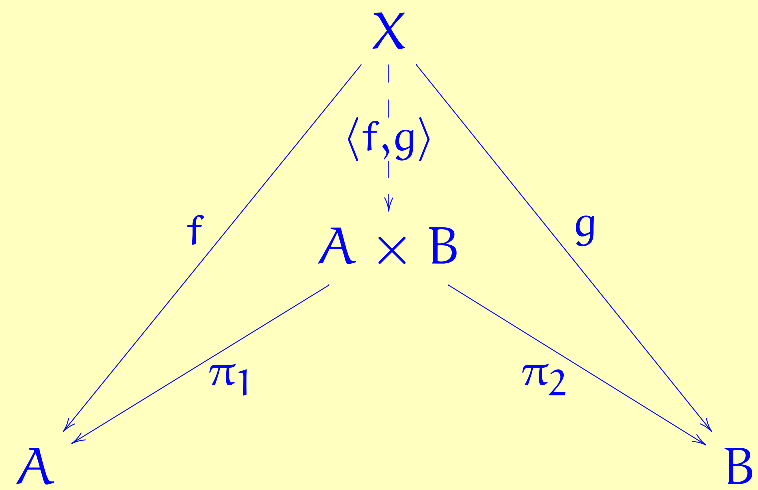
For all objects X and all morphisms $f : X \to A$ and $g : X \to B$, there exists a *unique* morphism $h : X \to C$ such that $f = \pi_1 \circ h$ and $g = \pi_2 \circ h$.



Categorical product, continued

Note: a categorical product, if it exists, is unique up to isomorphism.

Notation: we often write $C = A \times B$, $h = \langle f, g \rangle$.



Example: In **Set**, **Grp**, **Top**, **Vec**, **Pos**, categorical product is cartesian product (with the pointwise structure).

In a poset, categorical product is *meet*, i.e., *greatest lower bound*.

Products, continued

Proposition. In any category where they exist, categorical products satisfy

$$(A \times B) \times C \cong A \times (B \times C), \quad A \times B \cong B \times A$$

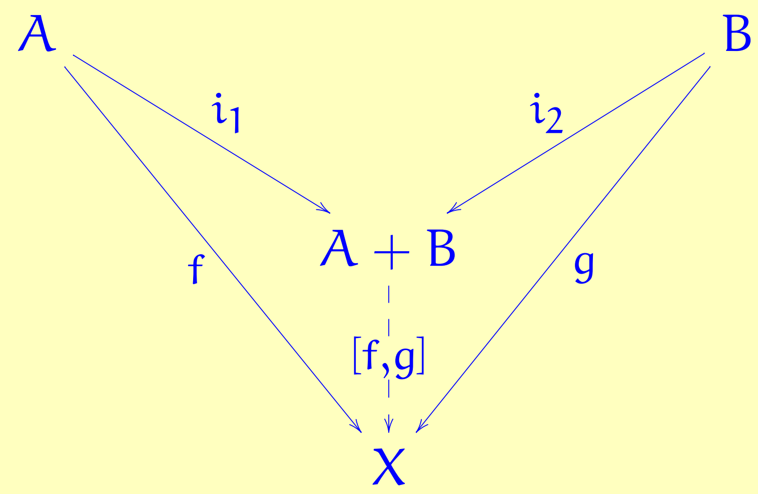
If 1 is a terminal object, then we also have

$$1 \times A \cong A \cong A \times 1.$$

Moreover, the above isomorphisms are natural.

Categorical coproduct

The dual concept of a product is a *coproduct*.

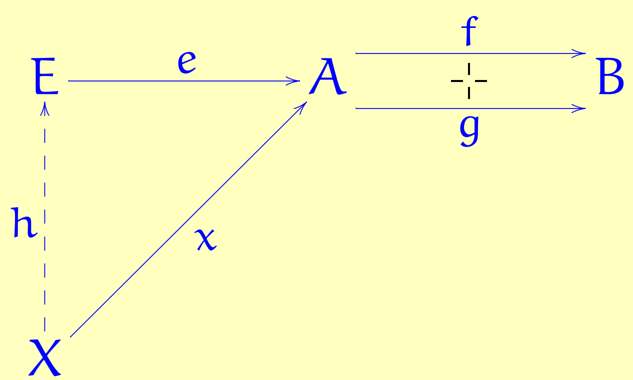


Example: In **Set**, coproduct is disjoint union. In **Vec** and **Ab** coproduct is direct sum. What is the coproduct, if any, in $\mathbf{Set}_\perp$?

Equalizers

Let $f, g : A \to B$ be morphisms in a category. An *equalizer* of f and g is a pair (E, e) where E is an object, $e : E \to A$ is a morphism, and such $f \circ e = g \circ e$, and such that the following property holds:

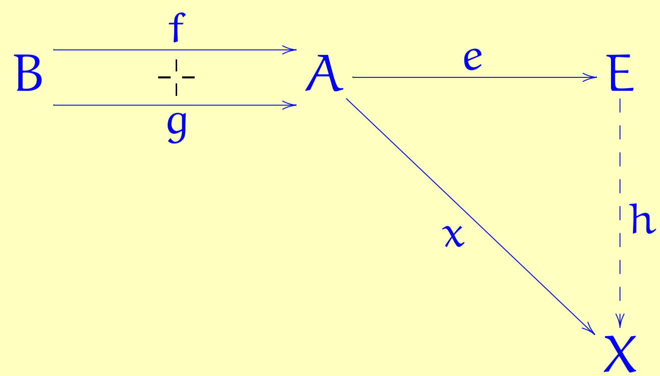
For all objects X and all morphisms $x : X \to A$ with $f \circ x = g \circ x$, there exists a *unique* morphism $h : X \to E$ such that $x = e \circ h$.



Example: in **Set**, an equalizer is the *graph of an equation*, i.e., $E = \{x \in A \mid f(x) = g(x)\}$.

Coequalizers

The dual concept of an equalizer is a coequalizer. In **Set**, this corresponds to the quotient of an *equivalence relation*.



Products and adjoints

Let $\mathbf{C}$ be a category with products. Consider the functors

$$F : \mathbf{C} \to \mathbf{C} \times \mathbf{C}, \quad G : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$$

given by

- $F(A) = (A, A)$ (and similarly on morphisms),
- $G(A, B) = A \times B$.

Then F is a *left adjoint* of G:

$$(\mathbf{C} \times \mathbf{C})((X, X), (A, B)) \cong \mathbf{C}(X, A \times B).$$

Concretely, this means that morphisms $(f, g) : (X, X) \to (A, B)$ in $\mathbf{C} \times \mathbf{C}$ are in natural bijective correspondence with morphisms $h : X \to A \times B$.

Moreover, because adjoints are unique, this property is *equivalent* to the definition of products.

Exponential objects

Let $\mathbf{C}$ be a category with products, and let A, B be objects. We say that the *exponential* of A and B exists if there is an object E and a natural isomorphism

$$\mathbf{C}(X \times A, B) \cong \mathbf{C}(X, E).$$

(natural in X). In this case, we usually write $E = B^A$, so that:

$$\mathbf{C}(X \times A, B) \cong \mathbf{C}(X, B^A).$$

Informally, B^A is a *space of functions* from A to B.

In other words, there is a bijective correspondence between morphisms $f : X \times A \to B$ and morphisms $f : X \to B^A$.

Exponential objects, continued

The definition of exponential objects can be understood in several equivalent ways.

More abstractly: A is *exponentiable* if the functor $F(X) = X \times A$ has a *right adjoint*. In this case, the right adjoint is written $G(B) = B^A$.

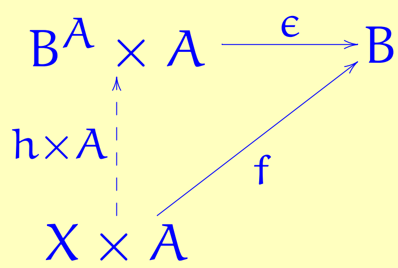
$$\mathbf{C}(X \times A, B) \cong \mathbf{C}(X, B^A).$$

$$\mathbf{C}(F(X), B) \cong \mathbf{C}(X, G(B)).$$

Exponential objects, continued

More concretely: An *exponential* for A and B is given by a pair (B^A, ϵ) where B^A is an object, $\epsilon : B^A \times A \to B$ is a morphism, and such that the following property holds:

For any object X and morphism $f : X \times A \to B$, there exists a *unique* morphism $h : X \to B^A$ such that



Exponential objects, if they exist, are unique up to isomorphism. We write $h = f^*$.

Cartesian-closed categories

Definition. A *cartesian-closed category* is a category with finite products (i.e., a products and a terminal object) and with exponential objects.

Part III: Lambda calculus

Recall: Types in programming

In computing, the *type* of a variable is the set of values that the variable can take. Examples of simple types are:

$$\mathbf{bit}, \mathbf{nat}, \mathbf{int}, \mathbf{string}, \ldots$$

We write $x : A$ to indicate that the variable x has type A.

Simple types can be combined by *type operations*. Examples:

$A \times B$:	Cartesian product (pairs of an A and a B)
$A + B$:	Disjoint union (either an A or a B)
$\mathbf{list}\, A$:	Type of lists of A's

We write (x, y) for a pair, and $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$ for the first and second component, respectively.

Higher-order functions

We write $f : A \to B$ for a *function* that takes inputs of type A and produces outputs of type B.

We can also regard $A \to B$ as a *type*, namely the type of all functions from A to B. This is called a *function space*.

A *higher order type* is a type where a function space occurs in a nested way, for example:

- a function that inputs another function: $(A \to B) \to C$,
- a function that outputs another function: $A \to (B \to C)$,
- a pair of two functions: $(A \to B) \times (C \to D)$.

A *higher order function* is a function of higher order type.

We need a language for manipulating higher order functions.

Example: Arithmetic expressions

Arithmetic expressions are made up from variables ($x, y, z \ldots$), numbers ($1, 2, 3, \ldots$), and operators ("$+$", "$-$", "$\times$" etc.)

The expression $x + y$ stands for the *result* of an addition (not an *instruction* to add, or the *statement* that something is being added).

We write

$$A = (x + y) \times z^2$$

One could write this as sequence of instructions:

let $w = x + y$, then let $u = z^2$, then let $A = w \times u$.

But such instructions would be cumbersome to manipulate, and algebraic laws impossible to state. Nested expressions are a powerful tool (which we take for granted).

Lambda calculus

The lambda calculus is an expression language for functions.
We normally write

Let f be the function defined by $f(x) = x^2$. Then consider $A = f(5)$,

In the lambda calculus we can just write

$$A = (\lambda x.x^2)(5).$$

The expression $\lambda x.x^2$ stands *for the function* that maps x to x^2 (as opposed to the *instruction* of squaring x, or the *statement* that x is being squared).

As for arithmetic, some of the power of the notation derives from the ability to nest expressions.

Examples

The composition operation $\circ$ of two functions:

We can write $f \circ g$ as $\lambda x.f(g(x))$.

We can write $C(f, g) = f \circ g$ as

$$\lambda f.\lambda g.f \circ g = \lambda f.\lambda g.\lambda x.f(g(x)).$$

Here, if $f : A \to B$ and $g : B \to C$, then $f \circ g : A \to C$, so the type of C is

$$C : (A \to B) \to ((B \to C) \to (A \to C))$$

Examples

The function mappair takes a function f and a pair (x, y), and returns $(f(x), f(y))$. It is an example of a higher-order function.

$$\text{mappair} = \lambda f.\lambda p.(f(\pi_1 p), f(\pi_2 p)),$$
$$\text{mappair} : (A \to B) \to ((A \times A) \to (B \times B)).$$

Some do-it-yourself examples

Find lambda terms of the following types:

- $A \to A \times A$,
- $B \to (A \to A \times B)$,
- $(A \to C) \to (A \times B \to C)$,
- $A \to A \times B$,
- $(A \times (A \to B)) \to B$.

The Curry-Howard isomorphism

There is a fundamental connection between typed lambda calculus and intuitionistic propositional logic.

Translation:

- Basic types A, B, C are *propositional symbols*.
- Type operations $\times$, $+$, and $\rightarrow$ are *logical connectives* **and**, **or**, and $\Rightarrow$, respectively.

Proposition (Curry-Howard isomorphism): There exists a closed lambda term of a given type if and only if that type corresponds to a *tautology* of intuitionistic logic. Moreover, lambda terms correspond to *proofs*.

Examples

- $A \Rightarrow A$ **and** A — Provable: $\lambda x^A.(x, x)$
- $B \Rightarrow (A \Rightarrow A$ **and** $B)$ — Provable: $\lambda x^B.\lambda y^A.(y, x)$
- $(A \Rightarrow C) \Rightarrow (A$ **and** $B \Rightarrow C)$ — Provable: $\lambda f^{A \Rightarrow C}.\lambda p^{A \textbf{and} B}.f(\pi_1(p))$
- $A \Rightarrow A$ **and** B — Not provable.
- $(A$ **and** $(A \Rightarrow B)) \Rightarrow B$ — Provable: $\lambda x.(pi_2(x)(\pi_1(x)))$.

Cf. the *Brower-Heyting-Kolmogorov interpretation*: a proof of A **and** B is a pair of a proof of A and a proof of B. A proof of $A \Rightarrow B$ is a function that maps proofs of A to proofs of B.

The inference rules of intuitionistic logic

Assertions ("sequents") are of the form:

$$A_1, \ldots, A_n \vdash B,$$

meaning B is provable from assumptions $A_1, \ldots, A_n$.

$$\frac{}{\Gamma, A \vdash A} \qquad \frac{}{\Gamma \vdash \mathrm{T}}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \textbf{ and } B} \qquad \frac{\Gamma \vdash A \textbf{ and } B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \textbf{ and } B}{\Gamma \vdash B}$$

The typing rules of simply-typed lambda calculus

Assertions ("judgments") are of the form:

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : B,$$

meaning term M, with free variables $x_1, \ldots, x_n$ of respective types $A_1, \ldots, A_n$, is well-typed of type B.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash * : 1}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1(M) : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2(M) : B}$$

The evaluation of lambda terms

The basic computational rule of lambda calculus is β-reduction, which means, applying a function to an argument:

$$(\lambda x.M)N \to M[N/x],$$

$$\pi_1(M, N) \to M, \quad \pi_2(M, N) \to N.$$

We close these rules using *transitivity*, *reflexivity*, and *congruence*.

Theorem (Normalization): Every simply-typed lambda term reduces in a finite number of steps to a unique normal form.

Theorem (Subject reduction): If $\Gamma \vdash M : A$ is well-typed and $M \to^* N$, then $\Gamma \vdash N : A$.

With this, the lambda calculus is a (simple) programming language — see Lisp, ML, Haskell for real world examples.

The theory of $\beta\eta$ conversion

It makes sense to consider two lambda terms *equal* if they have the same normal form. Define β-equivalence, in symbols $=_\beta$ to the the smallest congruence relation containing β-reduction.

It also makes sense to consider so-called η-rules:

$$\begin{array}{ll} \lambda x.(Mx) =_\eta M, & \text{where } x \text{ is not free in } M, \\ (\pi_1 M, \pi_2 M) = M, & \text{where } M : A \times B, \\ * = M, & \text{where } M : 1. \end{array}$$

Let $=_{\beta\eta}$ be the smallest congruence relation containing β-reduction and η-equivalences.

The interpretation of simply-typed lambda calculus in Set

The simple type system can be interpreted in set theory, where a *type* is identified with a *set*.

- Basic types are interpreted as specific sets: $[\![\mathbf{bit}]\!] = \{0, 1\}$, $[\![\mathbf{nat}]\!] = \mathbb{N}$, etc.
- Type operations are interpreted as set operations:

$$\begin{aligned} [\![1]\!] &= 1, \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!], \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]}. \end{aligned}$$

- A context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is interpreted as a set:

$$[\![\Gamma]\!] = [\![A_1]\!] \times \ldots \times [\![A_n]\!].$$

- A typing judgement $\Gamma \vdash M : B$ is interpreted as a function:

$$[\![\Gamma \vdash M : B]\!] : [\![\Gamma]\!] \to [\![B]\!]$$

  defined *by recursion* on M.

The interpretation of simply-typed lambda calculus in cartesian-closed categories [Lambek]

Instead of *sets*, one can use the objects of any *cartesian-closed category*.

- Basic types are interpreted as specific *objects* $[\![A]\!]$, $[\![B]\!]$, etc.
- Type operations are interpreted *using the cartesian-closed structure*:

$$\begin{aligned} [\![1]\!] &= 1, \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!], \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]}. \end{aligned}$$

- A context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is interpreted as an *object*:

$$[\![\Gamma]\!] = [\![A_1]\!] \times \ldots \times [\![A_n]\!].$$

- A typing judgement $\Gamma \vdash M : B$ is interpreted as a *morphism*:

$$[\![\Gamma \vdash M : B]\!] : [\![\Gamma]\!] \to [\![B]\!]$$

defined *by recursion* on M.

The interpretation of simply-typed lambda calculus in cartesian-closed categories, continued

$$
\begin{array}{lcl}
[\![\Gamma, x : A \vdash x : A]\!] & = & [\![\Gamma]\!] \times [\![A]\!] \xrightarrow{\pi_2} [\![A]\!] \\
[\![\Gamma \vdash * : 1]\!] & = & [\![\Gamma]\!] \xrightarrow{*} 1 \\
[\![\Gamma \vdash \lambda x.M : A \to B]\!] & = & [\![\Gamma]\!] \xrightarrow{[\![\Gamma, x:A \vdash M:B]\!]^*} [\![B]\!]^{[\![A]\!]} \\
[\![\Gamma \vdash MN : B]\!] & = & [\![\Gamma]\!] \xrightarrow{\langle [\![\Gamma \vdash M:A \to B]\!], [\![\Gamma \vdash N:A]\!] \rangle} [\![B]\!]^{[\![A]\!]} \times [\![A]\!] \xrightarrow{\epsilon} [\![B]\!]. \\
[\![\Gamma \vdash (M, N) : A \times B]\!] & = & [\![\Gamma]\!] \xrightarrow{\langle [\![\Gamma \vdash M:A]\!], [\![\Gamma \vdash N:B]\!] \rangle} [\![A]\!] \times [\![B]\!] \\
[\![\Gamma \vdash \pi_1(M) : A]\!] & = & [\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash M:A \times B]\!]} [\![A]\!] \times [\![B]\!] \xrightarrow{\pi_1} [\![A]\!] \\
[\![\Gamma \vdash \pi_2(M) : B]\!] & = & [\![\Gamma]\!] \xrightarrow{[\![\Gamma \vdash M:A \times B]\!]} [\![A]\!] \times [\![B]\!] \xrightarrow{\pi_2} [\![B]\!]
\end{array}
$$

Theorem 1. The interpretation of the simply-typed lambda calculus in cartesian-closed categories is *sound*. In other words, if $\Gamma \vdash M = N : A$, then $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!]$. (Easy, by induction).

Theorem 2. The interpretation of the simply-typed lambda calculus in cartesian-closed categories is *complete*. In other words, if $[\![\Gamma \vdash M : A]\!] = [\![\Gamma \vdash N : A]\!]$ for *all* interpretations in *all* cartesian-closed categories, then $\Gamma \vdash M = N : A$.

Theorem 3. The simply-typed lambda calculus is an *internal language* for cartesian-closed categories.

(Roughly: there is a one-to-one correspondence between models of the lambda calculus and cartesian-closed categories).

The term model

Fix a set of basic types. The *term model* of the simply-typed lambda calculus is a category Λ, constructed as follows:

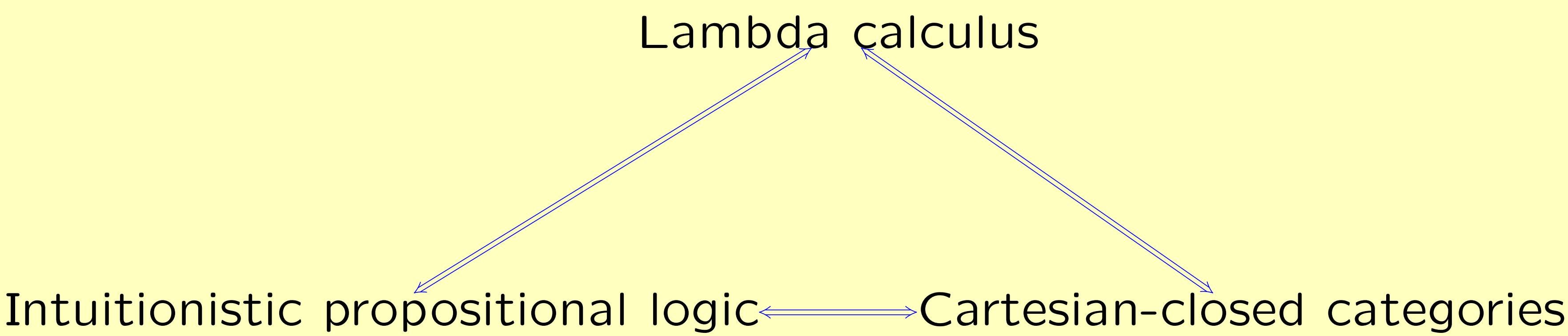
- The *objects* are types.
- A *morphism* $f : A \to B$ is a $\beta\eta$-equivalence class of typing judgements of the form $x : A \vdash M : B$.

Theorem 4. The term model Λ is a cartesian-closed category. Moreover, for any cartesian-closed category $\mathbf{C}$, there is a bijective correspondence between:

- Maps assigning objects of $\mathbf{C}$ to basic types;
- Interpretations of the lambda calculus in $\mathbf{C}$; and
- Cartesian closed functors $F : \Lambda \to \mathbf{C}$.

The Curry-Howard-Lambek isomorphism

By the results of the previous slides Λ is the *free* cartesian-closed category, and cartesian-closed categories and the lambda calculus (and therefore intuitionistic propositional logic) are essentially the same.



Extensions of the Curry-Howard isomorphism

The Curry-Howard isomorphism gives a basic connection between *programming languages* and *logic*. This connection can be usefully extended in both directions:

- given a programming language feature, one can ask for its logical meaning.
- Given a logical feature, one can ask for its computational meaning.

Examples:

Logic	Programming
A **or** B	Sum type $A + B$
$\forall$ quantifier	Polymorphism: $\lambda x.x : \forall A.A \to A$
$\exists$ quantifier	Data abstraction: $\exists D.(A \times D \to B) \times D$
Classical logic A **or** $\neg A$	Continuations
Type theory	Dependently typed programming
Topos logic	Set comprehension
...	...