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?

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

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

- $\mathrm{id}_\perp(a) = a$ (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

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

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

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

How do probabilistic programs compose?

- $\mathsf{id}_{\mathrm{pr}}(a)(b) = \begin{cases} 1 & \text{if } a = b \\ 0 & \text{otherwise} \end{cases}$ (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: $\text{"ab"} \cdot \text{"bc"} = \text{"abbc"}$.

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

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

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

How do such programs compose?

- $\mathsf{id}_{\mathrm{out}}(a) = (a, \epsilon)$ (the identity function writes nothing)
- $(g \circ_{\mathrm{out}} f)(a) = (c, s \cdot t)$ 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:

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

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

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

Such a structure $(T, \mathrm{id}_T, \circ_T)$ 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 $f : A \to B$ to indicate $f \in \mathbf{C}(A, B)$.

- with *operations*

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

  Note: this notation just means:

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

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

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

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 $\eta_A : \mathscr{P}^{\text{fin}}(A) \to A^*$ 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

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

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.