RILEY SHAHAR

TITLE

Contents

List of Tables

List of Figures

1 *Cryptography*

Foundations: Computational Hardness

2 Category Theory

The notion of a *category*, originally developed as an abstraction for certain ideas in pure mathematics, turns out to be the natural algebraic axiomatization of a collection of strongly typed, composable processes, such as functions in a strongly typed programming language. In this section, we will develop the basic theory of categories, prioritizing examples from computer science where possible[1].

[1] Basic texts on category theory include [Mac71] and [Rie17], while the connection to computer science is explored in [Pie91] and [BW90]. A more advanced treatment of the connection, especially applications to programming language theory, is [Jac99].

Categories

Definition 2.1 (Category)**.** A *category* $\mathcal{C}$ consists of the following data:

- a collection[2] of objects, overloadingly also called $\mathcal{C}$;
- for each pair of objects $x, y \in \mathcal{C}$, a collection of *morphisms* $\mathcal{C}(x, y)$;
- for each object $x \in \mathcal{C}$, a designated *identity morphism* $x \xrightarrow{1_x} x$;
- for each pair of morphisms $x \xrightarrow{f} y \xrightarrow{g} z$, a designated *composite morphism* $x \xrightarrow{gf} z$.

[2] We use the word *collection* for foundational reasons: in many important examples, the objects and morphisms do not form sets. We ignore such foundational issues here; they are discussed in [Mac71, Section 1.6].

This data must satisfy the following axioms:

- *unitality*: for any $x \xrightarrow{f} y$, $1_y f = f = f 1_x$;
- *associativity*: for any $x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} w$, $(hg)f = h(gf)$.

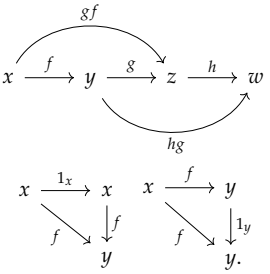

Figure 2.1: The category axioms.

Notation. In addition to those used above, many syntaxes are common for basic categorical notions.

- A morphism $f \in \mathcal{C}(x, y)$ is often written $f \colon x \to y$ or $x \xrightarrow{f} y$; x is called its *domain* or *source* and y is called it *codomain* or *target*.
- Morphisms may be called maps, arrows, or homomorphisms; the class of morphisms $\mathcal{C}(x, y)$ may also be written $\mathrm{Hom}_{\mathcal{C}}(x, y)$ or just $\mathrm{Hom}(x, y)$.
- Composition is written gf or $g \circ f$, or sometimes in the left-to-right order fg.
- Identities are written 1_x, id_x, or just x where the context is clear[3].

[3] I agree with Harold Simmons, who says that this last is "a notation so ridiculous it should be laughed at in the street" [Sim11, p. 5].

Example 2.2 (Functional Programming Languages)**.** Consider some strongly-typed functional programming language L, whose functions

are never side-effecting. Then under very modest assumptions about L, we can make a category $\mathcal{L}$, as follows:

- the objects of $\mathcal{L}$ are the types of L;
- the morphisms $\mathcal{L}(A, B)$ are the functions of type $A \to B$;
- the identities 1_A are the identity functions $\lambda(x : A).x$;
- composition of morphisms are the usual function composition.

If L is truly non-side-effecting, then it's straightforward to check that this construction does indeed satisfy the axioms of a category; see for instance [BW90, Section 2.2] to see the necessary assumptions spelled out rigorously.

Categories are also widespread in mathematics, as the following examples show.

Example 2.3 (Concrete Categories)**.** The following are all categories:

- SET is the category of sets and functions.
- GRP is the category of groups and group homomorphisms.
- RING is the category of rings and ring homomorphisms.
- TOP is the category of topological spaces and homeomorphisms.
- For any field $\Bbbk$, $\text{VECT}_\Bbbk$ is the category of vector spaces over $\Bbbk$ and linear transformations.

We call such categories, whose objects are structured sets and whose morphisms are structure-preserving set-functions, *concrete*. On the other hand, many categories look quite different.

Example 2.4. The following are also categories:

- The *empty category* has no objects and no morphisms.
- The *trivial category* has a single object and its identity morphism.
- Any group (or, more generally, monoid) can be thought of as a category with a single object, a morphism for every element, and composition given by the monoid multiplication.
- Any poset (or, more generally, preorder) $(P, \leq)$ can be thought of as a category whose objects are the elements of P, with a unique morphism $x \to y$ if and only if $x \leq y$. In this sense, composition is a "higher-dimensional" transitivity, and identities are higher-dimensional reflexivity.
- Associated to any directed graph is the *free category* on the graph, whose objects are nodes and whose morphisms are paths. In particular, the identities are just the empty paths, while composition concatenates two paths.
- There is a category whose objects are (roughly) multisets of molecules and whose morphisms are chemical reactions. See [BP17] for a formalization of this notion.

When working with categories, we often want to show that two complex composites equate. In this case, we prefer graphical notation to the more traditional symbolic equalities of Definition 2.1. The key idea is that such diagrams can be "pasted", allowing us to build up complex equalities from simpler ones.

Definition 2.5 (Commutative Diagram)**.** A diagram[4] *commutes* if, for any pair of paths through the diagram with the same start and end, the composite morphisms are equal.

[4] The notion of a diagram can be made precise fairly easily; see [Rie17, Section 1.6].

Example 2.6. In this language, the axioms of Definition 2.1 are expressed by commutativity of the diagrams in Figure 2.1.

Functors

Bibliography

[And17] Scott Anderson. "P $\stackrel{?}{=}$ NP". In: (2017). URL: https://www.scottaaronson.com/papers/pnp.pdf.

[BK22] Anne Broadbent and Martti Karvonen. "Categorical composable cryptography". In: *Foundations of software science and computation structures*. Vol. 13242. Lecture Notes in Comput. Sci. Springer, Cham, 2022, pp. 161–183. ISBN: 9783030992538. DOI: 10.1007/978-3-030-99253-8_9. URL: https://doi.org/10.1007/978-3-030-99253-8_9.

[BP17] John C. Baez and Blake S. Pollard. "A compositional framework for reaction networks". In: *Reviews in Mathematical Physics* 29.09 (Sept. 2017), p. 1750028. DOI: 10.1142/s0129055x17500283. URL: https://doi.org/10.1142%2Fs0129055x17500283.

[BW90] Michael Barr and Charles Wells. *Category theory for computing science*. USA: Prentice-Hall, Inc., 1990. ISBN: 0131204866.

[Can00] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Paper 2000/067. https://eprint.iacr.org/2000/067. 2000. URL: https://eprint.iacr.org/2000/067.

[Can08] Ran Canetti. *Lecture 11*. Spring 2008. URL: https://www.cs.tau.ac.il/~canetti/f08-materials/scribe11.pdf.

[Can20] Ran Canetti. "Universally Composable Security". In: *J. ACM* 67.5 (Sept. 2020). ISSN: 0004-5411. DOI: 10.1145/3402457. URL: https://doi.org/10.1145/3402457.

[CF01] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Joe Kilian. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 19–40. ISBN: 978-3-540-44647-7.

[CFS16] Bob Coecke, Tobias Fritz, and Robert W. Spekkens. "A mathematical theory of resources". In: *Information and Computation* 250 (2016). Quantum Physics and Logic, pp. 59–86. ISSN: 0890-5401. DOI: https://doi.org/

10.1016/j.ic.2016.02.008. URL: https://www.sciencedirect.com/science/article/pii/S0890540116000353.

[GK96] Oded Goldreich and Hugo Krawczyk. "On the Composition of Zero-Knowledge Proof Systems". In: *SIAM Journal on Computing* 25.1 (1996), pp. 169–192. DOI: 10.1137/S0097539791220688. eprint: https://doi.org/10.1137/S0097539791220688. URL: https://doi.org/10.1137/S0097539791220688.

[GL89] O. Goldreich and L. A. Levin. "A Hard-Core Predicate for All One-Way Functions". In: *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*. STOC '89. Seattle, Washington, USA: Association for Computing Machinery, 1989, pp. 25–32. ISBN: 0897913078. DOI: 10.1145/73007.73010. URL: https://doi.org/10.1145/73007.73010.

[Gol01] O. Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools*. Cambridge University Press, 2001. ISBN: 9780521791724.

[HM03] Dennis Hofheinz and Jörn Müller-Quade. *A Paradox of Quantum Universal Composability*. 2003. URL: https://www.quiprocone.org/Hot%20Topics%20posters/muellerquade_poster.pdf.

[Jac99] B. Jacobs. *Categorical Logic and Type Theory*. Studies in Logic and the Foundations of Mathematics 141. Amsterdam: North Holland, 1999.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC Cryptography and Network Security Series. Taylor & Francis, 2014. ISBN: 9781466570269.

[Lin17] Yehuda Lindell. "How to Simulate It—A Tutorial on the Simulation Proof Technique". In: *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. Springer International Publishing, 2017, pp. 277–346. ISBN: 9783319570488. DOI: 10.1007/978-3-319-57048-8_6.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Vol. 5. New York: Springer-Verlag, 1971, pp. ix+262.

[MR92] Silvio Micali and Phillip Rogaway. "Secure Computation". In: *Advances in Cryptology — CRYPTO '91*. Ed. by Joan Feigenbaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 392–404. ISBN: 978-3-540-46766-3.

[Ped91] Torben Pryds Pedersen. "Non-interactive and information-theoretic secure verifiable secret sharing". In: *Annual international cryptology conference*. Springer. 1991, pp. 129–140.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, Aug. 1991. ISBN: 9780262288460. DOI: 10.7551/mitpress/1524.001.0001. URL: https://doi.org/10.7551/mitpress/1524.001.0001.

[PS10] Raphael Pass and Abhi Shelat. *A Course in Cryptography*. 2010. URL: https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf.

[Rie17] Emily Riehl. *Category theory in context*. en. Courier Dover Publications, Mar. 2017.

[Ros21] Mike Rosulek. *The Joy of Cryptography*. 2021. URL: https://joyofcryptography.com.

[Sim11] Harold Simmons. *An Introduction to Category Theory*. USA: Cambridge University Press, 2011. ISBN: 0521283043.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. ISBN: 113318779X.

[Tre09] Luca Trevisan. *Notes for Lecture 27*. Apr. 2009. URL: https://theory.stanford.edu/~trevisan/cs276/lecture27.pdf.

[Unr10] Dominique Unruh. "Universally Composable Quantum Multi-Party Computation". In: *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*. EUROCRYPT'10. French Riviera, France: Springer-Verlag, 2010, pp. 486–505. ISBN: 3642131891. DOI: 10.1007/978-3-642-13190-5_25. URL: https://doi.org/10.1007/978-3-642-13190-5_25.

[Yao82] Andrew C. Yao. "Theory and application of trapdoor functions". In: *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. 1982, pp. 80–91. DOI: 10.1109/SFCS.1982.45.

Theory

In the main body, we have assumed standard material from a course in computability and complexity, including function asymptotics, the notion of an algorithm, and the complexity class P. We briefly overview these ideas here; a standard text is [Sip13].

Asymptotics

Function asymptotics formalize the notion of a function approximating another function. In particular, for a pair of functions $f, g : \mathbb{N} \to \mathbb{R}$, we often want to compare f and g on large inputs and only up to a constant factor. This is most common in runtime analysis, the idea being that the running time of algorithms on small inputs is less important to their overall performance than their running time on large inputs. We formalize this notion as follows:

Definition .7 (Function Asymptotics)**.** Let $f, g : \mathbb{N} \to \mathbb{R}$ be a pair of functions. We say that f is *big-Oh of* g, written $f = O(g)$, if there exists a constant $c > 0$ such that for all n sufficiently large,

$$cf(n) \geq g(n).$$

In this case, we also say that g is *big-Omega of* f, written $g = \Omega(f)$.

If $f = O(g)$ and $g = O(f)$, we say that f is *big-Theta of* g, written $f = \Theta(g)$. Explicitly, this means that there exist constants $c_1, c_2 > 0$ such that for all n sufficiently large,

$$c_1 f(n) \leq g(n) \leq c_2 f(n).$$

If $f = O(g)$ but $f \neq \Theta(g)$, we say that f is *little-oh of* g, written $f = o(g)$, and g is *little-omega of* f, written $g = \omega(f)$. Explicitly, this means that for all constants $\epsilon > 0$ and all n sufficiently large,

$$f(n) \leq \epsilon g(n).$$

Notation. By abuse of notation, we often write $f(n) = O(g(n))$ to mean that f is $O(n)$; for example, the statement that n^2 is $O(n^3)$ means that the function $f(n) = n^2$ is $O(g)$, where g is the function $n \mapsto n^3$.

Example .8. We have that:

- $17n^2$ is $o(n^3)$, $\omega(n)$, and $\Theta(n^2)$;
- $\log n$ is $o(n)$, $\omega(1)$, and $\Theta(\ln n)$;
- e^n is $\omega(n^k)$ for any exponent k;
- e^{-n} is $o(n^{-k})$ for any exponent k.

These last two examples are especially important. No matter how big the power, an exponential will always dominate a polynomial for sufficiently big n. Because of the importance of polynomials in theoretical computer science, we say a function is *negligible* if it is $o(n^k)$ for all k.

Proposition .9. *Big-Oh is a preorder on the set of functions $\mathbb{N} \to \mathbb{N}$. The induced equivalence relation is exactly big-Theta.*

In the partial order of equivalence classes under Θ, O behaves like $\leq$, o like $<$, and Θ like $=$. As suggested by the notation $f = O(g)$, it is common in some contexts to treat functions as identical with their asymptotic equivalence class.

Proposition .10. *Let $f_1 = O(g_1)$ and $f_2 = O(g_2)$. Let c be any nonzero constant. Then,*

$$f_1 + f_2 = O(\max\{g_1, g_2\}), \quad cf_1 = O(g_1), \quad \textit{and} \quad f_1 f_2 = O(g_1 g_2).$$

In other words,

$$O(g_1) + O(g_2) = O(\max\{g_1, g_2\}), \quad cO(g) = O(g), \quad \textit{and} \quad O(g_1)O(g_2) = O(g_1 g_2).$$

Identical results hold for o and Θ.

Proposition .10 justifies the universal practice of dropping constants and small additive terms from asymptotics, so that for instance $n^2 + n + \ln n = \Theta(n^2)$.

Algorithms and Determinism

Our basic notion is of an *algorithm* over a finite alphabet Σ, usually $\{0, 1\}$. An algorithm $\mathcal{A}$ is intuitively some set of steps which take an input word x over Σ, perform some transformations, and output another word $\mathcal{A}(x)$ over Σ. An algorithm may have certain *side effects*, such as sending a message or logging a string, and its behavior may not be deterministic. There are several ways to formalize the notion of algorithm—most common in cryptography are Turing machines—but we will not need to be so precise here.

Algorithms may have multiple possible "branches" in their instructions. Consider the following:

Algorithm .11. On input x, either output 0 or 1.

We say that algorithms of this sort are *nondeterministic*; in contrast, an algorithm is *deterministic* if its instructions do not include such choices. In particular, we say that an algorithm $\mathcal{A}$ *deterministically computes* a function f if it is deterministic and, for any input $x \in \Sigma^*$, $\mathcal{A}$ outputs the value $f(x) \in \Sigma^*$. In contrast, $\mathcal{A}$ *nondeterministically computes* f if, for any input x, there exists a particular choice of branches such that $\mathcal{A}$ outputs $f(x)$. Thus Algorithm .11 nondeterministically computes both the functions $x \mapsto 0$ and $x \mapsto 1$. We sometimes view nondeterministic algorithms as computing functions into the power set of Σ^*, so that Algorithm .11 computes the function $x \mapsto \{0,1\}$, and we similarly sometimes write $\mathcal{A}(x) = \{0,1\}$.

An important middle ground is *probabilistic* algorithms. Again, there are many possible models, but the basic idea is that a probabilistic algorithm has access to some source of randomness—say, an arbitrarily long string of independent and uniform coin tosses—which it can use to choose between branches. In this case, it is not enough for there to be some branch which computes a specific function. Instead, we say that an algorithm *computes f with bounded probability* if for any input x,

$$\Pr[\mathcal{A}(x) = f(x)] > \frac{2}{3},$$

where the probability is taken over the randomness of $\mathcal{A}$[5]. In this case, we often think of $\mathcal{A}(x)$ as a probability distribution on Σ^*.

Instead of thinking of algorithms operating directly on binary strings, we usually think of them as operating on encodings of mathematical objects. For example:

Algorithm .12. On input x a natural number, output the number $2x$.

We say that Algorithm .12 deterministically computes $x \mapsto 2x$, even though it technically operates on encodings of naturals. While there are many possible encodings, we assume that a reasonable encoding is chosen, so that for instance numbers are encoded in binary, rather than unary. Such details will not be relevant for us.

One more subtlety is important. In general, we require that the description of any algorithm $\mathcal{A}$ is finite. However, we may also consider *non-uniform* algorithms, which are sequences of algorithms $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \ldots)$ such that, on an input of length n, $\mathcal{A}$ delegates to $\mathcal{A}_n$. Non-uniform computation is generally stronger than uniform computation, as non-uniform algorithms may encode nonfinite information, as long as they only use finitely much of this information for each input length and hence for each computation[6].

[5] The choice of $\frac{2}{3}$ is not particularly important here—generally any constant $c > \frac{1}{2}$ works.

[6] For instance, non-uniform algorithms may solve the halting problem (which asks whether an input algorithm $\mathcal{A}$ eventually terminates), which is uniformly undecidable. In particular, since there are only finitely many Turing machines of a given size, a non-uniform algorithm may simply encode in $\mathcal{A}_n$ the answer to the halting problem for each Turing machine of length n.

Complexity Theory

Each algorithm has an associated *running time*, which is informally the number of steps the algorithm takes on a given input. In particular, for an algorithm $\mathcal{A}$, we say that its running time is the function $T_{\mathcal{A}} : \mathbb{N} \to \mathbb{N}$ which takes any natural number n to the maximum number of steps $\mathcal{A}$ takes to terminate on any input of length n. Of course, this notion is not yet precise, as we don't know what a "step" is, but it is easy to make precise in any standard model of computation.

In general, the running time may depend on the formal model of computation in which the algorithm is constructed, but the *complexity-theoretic Church-Turing thesis* states that "reasonable" models of classical computation recover the same inhabitants of sufficiently robust complexity classes, in particular of those we are about to define. This hypothesis is a heuristic, but has been born out in practice.

Definition .13 (polynomial-time; P, NP)**.** An algorithm $\mathcal{A}$ is *polynomial-time* if $T_{\mathcal{A}} = O(n^k)$ for some constant k. The class P consists of all functions which are deterministically computable by polynomial-time algorithms. The class NP consists of all functions which are nondeterministically computable by polynomial-time algorithms[7].

The general idea is that polynomial-time algorithms "efficient in practice." In may sometimes occur that the constant factors or the exponent are so large as to render the algorithm practically useless, but in We can now state the most important open problem in computer science:

Conjecture .14. *We have that $P \neq NP$.*

While a proof seems completely out of reach, this conjecture is widely believed, and as we will see is necessary for all of modern cryptography; we will assume it here. An introduction to the modern state of P vs. NP is [And17].

Formalizing probabilistic complexity classes is slightly more subtle. Consider the following case:

Algorithm .15. On input x, output 1 with probability $1 - 2^{-|x|}$; otherwise count from 0 to $2^{|x|}$ and then output 1.

While this algorithm is almost always polynomial-time, it is not polynomial-time when it takes the second branch. The point is that for probabilistic algorithms, $T_{\mathcal{A}}(n)$ is a probability distribution, not just a fixed number. For our purposes, we require that the algorithm *always* runs in polynomial time. As such:

Definition .16 (probabilistic polynomial-time; BPP)**.** A probabilistic algorithm is *probabilistic polynomial-time* if, for any choice of random

[7] In fact, we have defined here the classes FP and FNP of polynomially- and nondeterministically-polynomially-computable *function problems*. Formally, P and NP are classes of *decision problems*, which are just subsets L of Σ^*—the algorithm must output 1 if its input is in L, and 0 otherwise. Function and decision problems are extremely closely related—for instance, P = NP if and only if FP = FNP—and we will not distinguish between them here.

bits, $T_{\mathcal{A}} = O(n^k)$ for some constant k. The class BPP consists of all functions which are computable with bounded probability by a probabilistic polynomial-time algorithm.

For non-uniform algorithms, the situation is also slightly more complicated. In particular, it is too much to allow the machines to be arbitrarily large, as they could simply encode lookup tables for every possible input. As such, we ask that the size of each machine is polynomially bounded.

Definition .17 (non-uniform polynomial-time; P/Poly)**.** A non-uniform algorithm $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2, \dots)$ is *polynomial-time* if $T_{\mathcal{A}} = O(n^k)$ for some constant k and the size of each $\mathcal{A}_n$ is $O(n^k)$ for some constant k independent of n. The class P/poly consists of all functions which are computable by non-uniform polynomial-time algorithms.

Non-uniform probabilistic algorithms are similarly defined. We have the following important result:

Theorem .18 (Adelman's theorem)**.** *We have that BPP $\subseteq$ P/poly.*