Category Theory and Functional Programming
Course Introduction

Andrés Sicard-Ramírez

Universidad EAFIT

Semester 2022-2

Pedagogical Pact

Course web page

https://asr.github.io/category-theory/

Exams, text book, programming labs, etc.

See course web page.

Evaluación a la docencia

La evaluación a la docencia es obligatoria

Preliminaries

Convention

The number assigned to chapters, examples, exercises, figures, pages, sections, and theorems on these slides correspond to the numbers assigned in the textbook [Abramsky and Tzevelekos 2011].

About Category Theory

About Category Theory

Category theory as the essence of composition

In category theory we shall talk about objects and arrows between them.

About Category Theory

Category theory as the essence of composition

In category theory we shall talk about objects and arrows between them.

$$A \xrightarrow{f} B$$

About Category Theory

Category theory as the essence of composition

In category theory we shall talk about objects and arrows between them.

$$A \xrightarrow{f} B$$

subject	objects	arrows
set theory	sets	functions
logic	propositions	conditional proofs
programming languages	types	programs
quantum mechanics	systems	processes

About Category Theory

Informal description

In the introduction of their article, Abramsky and Tzevelekos [2011, p. 4] wrote:

> *Category theory can be seen as a 'generalised theory of functions', where the focus is shifted from the pointwise, set-theoretic view of functions to an abstract view of functions as arrows.*

About Category Theory

Remark

The ‘modern’ definition of function is from Dirichlet who in 1837 wrote: ‘If a variable y is so related to a variable x that whenever a numerical value is assigned to x, there is a rule according to which a unique value of y is determined, then y is said to be a **function** of the independent variable x.’ [Merzcbach and Boyer 2011, p. 452].

About Category Theory

Informal description

In the introduction of his book, Awodey [2010, p. 1] wrote:

> *As a first approximation, one could say that category theory is the mathematical study of (abstract) algebras of functions [...] We think of the composition $g \circ f$ as a sort of 'product' of the functions f and g, and consider abstract 'algebras' of the sort arising from collections of functions.*

About Category Theory

Informal description

D. S. Scott [1980] wrote:

> *General category theory provides a much purer theory of functions than set theory. Category theory gives a theory of functions [arrows] under composition and is also a theory of types.*

About Category Theory

Beginning

In Stanford Encyclopedia of Philosophy's entry to Category Theory, Marquis [2021, § 2] wrote:

> *Categories, functors, natural transformations, limits and colimits appeared almost out of nowhere in a paper by Eilenberg & Mac Lane (1945) entitled 'General Theory of Natural Equivalences' [...] The central notion at the time, as their title indicates, was that of natural transformation. In order to give a general definition of the latter, they defined functor, borrowing the term from Carnap, and in order to define functor, they borrowed the word 'category' from the philosophy of Aristotle, Kant, and C. S. Peirce, but redefining it mathematically.*

About Category Theory

An approach

In the introduction to the nice tutorials in [Pitt, Abramsky, Poigné and Rydeheard 1986], it is pointed out that [Abramsky 1986, p. 4]:

> *Perhaps the aspect which will excite most comment is the use of functional programming to motivate category theory [. . .] The idea, following [D. S. Scott 1980], is that a category is viewed as a collection of types and typed functions, i.e. an abstract functional programming language. A great deal of special categorical structure [. . .] can be interpreted in the programming context; while Backus' arguments in favour of 'function-level reasoning' [Backus 1978] can be seen as a special case of 'Lawvere's program' of replacing sets and elements by functions [. . .]. However, it should certainly be emphasized that there is more to category theory than functional programming; most notably, there are universal constructions.*

About Category Theory

An approach

In the introduction to the nice tutorials in [Pitt, Abramsky, Poigné and Rydeheard 1986], it is pointed out that [Abramsky 1986, p. 4]:

> *Perhaps the aspect which will excite most comment is the use of functional programming to motivate category theory [...] The idea, following [D. S. Scott 1980], is that a category is viewed as a collection of types and typed functions, i.e. an abstract functional programming language. A great deal of special categorical structure [...] can be interpreted in the programming context; while Backus' arguments in favour of 'function-level reasoning' [Backus 1978] can be seen as a special case of 'Lawvere's program' of replacing sets and elements by functions [...]. However, it should certainly be emphasized that there is more to category theory than functional programming; most notably, there are universal constructions.*

A similar approach is also followed in [Poigné 1992].

An Application: Semantics of Programming Languages

An Application: Semantics of Programming Languages

Approaches

There are various approaches for defining the meaning of programming languages (that is, of programs written in them).

- Operational semantics
  The meaning of a programming language is defined via an abstract machine for it.

An Application: Semantics of Programming Languages

Approaches

There are various approaches for defining the meaning of programming languages (that is, of programs written in them).

- Operational semantics
  The meaning of a programming language is defined via an abstract machine for it.

- Axiomatic semantics
  The meaning of a term is exactly what can be proved about it in some programming logic.
  The meaning of a programming language is defined via axioms and inference rules.

An Application: Semantics of Programming Languages

Approaches

There are various approaches for defining the meaning of programming languages (that is, of programs written in them).

- Operational semantics
  The meaning of a programming language is defined via an abstract machine for it.
- Axiomatic semantics
  The meaning of a term is exactly what can be proved about it in some programming logic. The meaning of a programming language is defined via axioms and inference rules.
- Denotational semantics
  The meaning of a term is a mathematical object. The meaning of a programming language is defined via semantics domains and interpretation functions.

An Application: Semantics of Programming Languages

Remark

In relation to the semantics approaches, Scott wrote [Shustek 2022, p. 29]:

> *I would say today that axiomatic, denotational, operational semantics all meld together, and the question is to take which aspects you want for an analysis or a proof, or for giving the foundations for some kind of implementation. You choose what is appropriate for the thing you want to accomplish.*

An Application: Semantics of Programming Languages

Remark

Domain theory (see, e.g. [Mitchell 1996; Plotkin 1992]) and category theory are often used for defining denotational semantics of functional languages.

Reading and Exercises

Reading

Reading

Chapters 1 and 2 from [Milewski 2019].

Exercises

A category consists of objects and arrows (morphisms). Arrows can be composed, and the composition is associative. Every object has an identity arrow that serves as a unit under composition. (Milewski [2019, p. 8])

Exercise 1

Is the world-wide web a category in any sense? Are links morphisms? [Milewski 2019, Challenge 1.4.4].

Exercise 2

Is Facebook a category, with people as objects and friendships as morphisms? [Milewski 2019, Challenge 1.4.5].

Exercise 3

When is a directed graph a category? [Milewski 2019, Challenge 1.4.6].

Types

Types

Types

Mathematics

A **type** is a range of significance of a propositional function. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell 1938, Appendix B: The Doctrine of Types].

In modern terminology, Rusell's types are domains of propositional functions.

Types

Mathematics

A **type** is a range of significance of a propositional function. Let $\varphi(x)$ be a (unary) propositional function. The type of $\varphi(x)$ is the range within which x must lie if $\varphi(x)$ is to be a proposition [Russell 1938, Appendix B: The Doctrine of Types].

In modern terminology, Rusell's types are domains of propositional functions.

Example

Let $\varphi(x)$ be the propositional function 'x is a prime number'. Then $\varphi(x)$ is a proposition only when its argument is a natural number.

$$\varphi : \mathbb{N} \to \{\mathrm{False}, \mathrm{True}\}$$
$$\varphi(x) := x \text{ is a prime number.}$$

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

- Type systems can be use for reasoning (internal o externally) about programs.

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

- Type systems can be use for reasoning (internal o externally) about programs.
- In general, types are compositional.

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

- Type systems can be use for reasoning (internal o externally) about programs.
- In general, types are compositional.
- Types systems are conservative.

Types

Programming languages

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])

- Type systems can be use for reasoning (internal o externally) about programs.
- In general, types are compositional.
- Types systems are conservative.
- Type systems can be use for facilitating large-scale software composition (handling higher-level modularity and user-defined abstractions).

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

- Type systems can be use for reasoning (internal o externally) about programs.
- In general, types are compositional.
- Types systems are conservative.
- Type systems can be use for facilitating large-scale software composition (handling higher-level modularity and user-defined abstractions).
- Type systems are good for detecting errors.

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

- Type systems can be use for reasoning (internal o externally) about programs.
- In general, types are compositional.
- Types systems are conservative.
- Type systems can be use for facilitating large-scale software composition (handling higher-level modularity and user-defined abstractions).
- Type systems are good for detecting errors.
- Types systems can be use for improving efficiency.

Types

Programming languages

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. (Pierce [2002, p. 1])*

- Type systems can be use for reasoning (internal o externally) about programs.
- In general, types are compositional.
- Types systems are conservative.
- Type systems can be use for facilitating large-scale software composition (handling higher-level modularity and user-defined abstractions).
- Type systems are good for detecting errors.
- Types systems can be use for improving efficiency.
- Types are useful for documentation (reading programs).

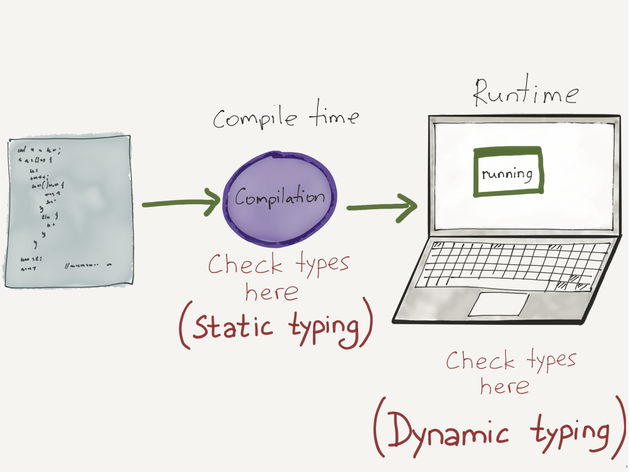
Types

Example

Examples of types in programming languages include integers, booleans, floating point numbers, characters, strings, lists, Cartesian products (tuples), discriminated unions, sets, functions, recursive/inductive types and user-defined types.

Types

Type checking: static typing vs dynamic typing†



†Figure from en.hexlet.io/courses/intro_to_programming/lessons/types/theory_unit.

Types

The static programmer says:

'Static typing catches bugs with the compiler and keeps you out of trouble.'

'Static languages are easier to read because they're more explicit about what the code does.'

'At least I know that the code compiles.'

'I trust the static typing to make sure my team writes good code.'

'Debugging an unknown object is impossible.'

'Compiler bugs happen at midmorning in my office; runtime bugs happen at midnight for my customers.'

The dynamic programmer says:

'Static typing only catches some bugs, and you can't trust the compiler to do your testing.'

'Dynamic languages are easier to read because you write less code.'

'Just because the code compiles doesn't mean it runs.'

'The compiler doesn't stop you from writing bad code.'

'Debugging overly complex object hierarchies is unbearable.'

'There's no replacement for testing, and unit tests find more issues than the compiler ever could.'

(From www.smashingmagazine.com/2013/04/introduction-to-programming-type-systems).

Types

Example
Dynamically typed: JavaScript, PHP and Python
Statically typed: C, C++, C#, Haskell, Java and Standard ML

Types

The propositions-as-types principle (Curry-Howard correspondence)

Types

The propositions-as-types principle (Curry-Howard correspondence)

Wadler [2015] introduces correspondence's levels by:

(i) Propositions as types

   'For each proposition in the logic there is a corresponding type in the programming language—and vice versa.'

Types

The propositions-as-types principle (Curry-Howard correspondence)

Wadler [2015] introduces correspondence's levels by:

(i) Propositions as types

‘For each proposition in the logic there is a corresponding type in the programming language—and vice versa.’

(ii) Proofs as programs

‘For each proof of a given proposition, there is a program of the corresponding type—and vice versa.’

Types

The propositions-as-types principle (Curry-Howard correspondence)

Wadler [2015] introduces correspondence's levels by:

(i) Propositions as types

‘For each proposition in the logic there is a corresponding type in the programming language—and vice versa.’

(ii) Proofs as programs

‘For each proof of a given proposition, there is a program of the corresponding type—and vice versa.’

(iii) Simplification of proofs as evaluation of programs

‘For each way to simplify a proof there is a corresponding way to evaluate a program—and vice versa.’

Types

Example

Example for the propositions-as-types principle.

(implication)	$A \supset B$	$\sigma \to \tau$	(function type)
(conjunction)	$A \wedge B$	$\sigma \times \tau$	(product type)
(disjunction)	$A \vee B$	$\sigma + \tau$	(sum type)
(bottom)	$\bot$	N_0	(empty type)
(top)	$\top$	N_1	(unit type)

Types

Homotopy type theory

See [Gonthier 2022, min 44:05].

Types

Lifted sets

Types

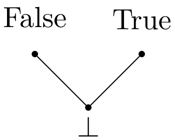
Lifted sets

Let A be a set. The lifted set $A_\perp$ is the poset with least element (bottom) $\perp$ and whose elements $A \cup \{\perp\}$ are ordered by

$$x \sqsubseteq y \quad \text{iff} \quad x = \perp \text{ or } x = y.$$

Example

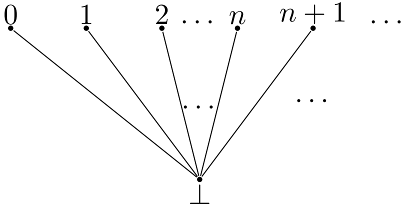
The lifted Booleans $B_\perp$.



Types

Example

The lifted natural numbers $N_\perp$.



Types

Haskell's types

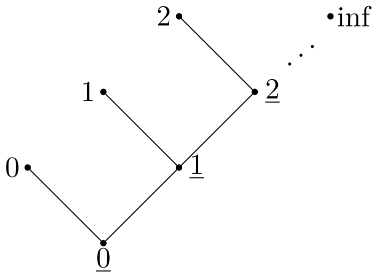
Haskell's types are lifted and lazy [Yang 2010].

Types

Example

Haskell's lazy natural numbers where $\texttt{Succ}\ \perp \neq \perp$ [Escardó 1993].

```
data Nat = Zero
         | Succ Nat
```



$$\underline{0} = \perp,$$
$$\underline{n+1} = \texttt{Succ}\ \underline{n},$$
$$\mathrm{inf} = \bigcup_{n \in \omega} \underline{n}.$$

Presentations

Presentations

Some possible topics

- Representation of number-theoretic functions in category theory [Lambek and P. J. Scott 1994, Part III].
- The categorical abstract machine [Cousineau, Curien and Mauny 1985, 1987].
- A categorical programming language [Hagino 1987a,b].
- Categorification
  - Introduction via examples [Lauda and Sussan 2022].
  - The graph minor category [Ramos 2022].
- Sets, types, categories and foundations of mathematics [Awodey 2011].

References

References

Abramsky, S. and Tzevelekos, N. (2011). Introduction to Categories and Categorical Logic. In: New Structures for Physics. Ed. by Coecke, B. Vol. 813. Lecture Notes in Physics. Springer, pp. 3–94. DOI: 10.1007/978-3-642-12821-9_1 (cit. on pp. 3, 8).

Abramsky, S. (1986). Introduction. In: Category Theory and Computer Programming. Ed. by Pitt, D., Abramsky, S., Poigné, A. and Rydeheard, D. Vol. 240. Lecture Notes in Computer Science. Springer-Verlag, pp. 3–5. DOI: 10.1007/3-540-17162-2 (cit. on pp. 13, 14).

Awodey, S. [2006] (2010). Category Theory. 2nd ed. Vol. 52. Oxford Logic Guides. Oxford University Press (cit. on p. 10).

— (2011). From Sets to Types, to Categories, to Sets. In: Foundational Theories of Classical and Constructive Mathematics. Ed. by Sommaruga, G. Vol. 76. The Western Ontario Series in Philosophy of Science. Springer, pp. 113–125. DOI: 10.1007/978-94-007-0431-2_5 (cit. on p. 53).

Backus, J. (1978). Can Programming Be Liberated from the von Neumann style? A Functional Style and its Algebra of Programs. Communications of the ACM 21.8, pp. 613–641. DOI: 10.1145/359576.359579 (cit. on pp. 13, 14).

References

Cousineau, G., Curien, P.-L. and Mauny, M. (1985). The Categorical Abstract Machine. In: Functional Programming Languages and Computer Architecture. Ed. by Jouannaud, J.-P. Vol. 201. Lecture Notes in Computer Science. Springer, pp. 50–64. DOI: 10.1007/3-540-15975-4_29 (cit. on p. 53).

— (1987). The Categorical Abstract Machine. Science of Computer Programming 8.2, pp. 173–202. DOI: 10.1016/0167-6423(87)90020-7 (cit. on p. 53).

Escardó, M. H. (1993). On Lazy Natural Numbers with Applications to Computability Theory and Functional Programming. SIGACT News 24.1, pp. 61–67. DOI: 10.1145/152992.153008 (cit. on p. 51).

Gonthier, G. (7th July 2022). Computer Proofs: Teaching Computers Mathematics, and Conversely. Conference in the International Congress of Mathematicians 2022. URL: https://opade.digital/days/2/sessions/-LF4j-pRKKgQBzQFnVf0Q (cit. on p. 45).

Hagino, T. (1987a). A Categorical Programming Language. PhD thesis. University of Edinburgh (cit. on p. 53).

References

Hagino, T. (1987b). A Typed Lambda Calculus with Categorical Type Constructors. In: Category Theory and Computer Programming. Ed. by Pitt David H. Poigné, A. and Rydeheard, D. E. Vol. 283. Lecture Notes in Computer Science. Springer, pp. 140–157. DOI: 10.1007/3-540-18508-9_24 (cit. on p. 53).

Lambek, J. and Scott, P. J. (1994). Introduction to Higher Order Categorical Logic. Cambridge University Press (cit. on p. 53).

Lauda, A. D. and Sussan, J. (2022). An Invitation to Categorification. Notices of the AMS 69.1, pp. 11–21. DOI: 10.1090/noti2399 (cit. on p. 53).

Marquis, J.-P. (2021). Category Theory. In: The Stanford Encyclopedia of Philosophy. Ed. by Zalta, E. N. Fall 2021. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/fall2021/entries/category-theory/ (visited on 16/06/2022) (cit. on p. 12).

Merzcbach, U. C. and Boyer, C. B. [1968] (2011). A History of Mathematics. 3rd ed. John Wiley & Sons (cit. on p. 9).

Milewski, B. (2019). Category Theory for Programmers. Version 32. 17 May 2022. URL: https://github.com/hmemcpy/milewski-ctfp-pdf (cit. on pp. 22, 23).

Mitchell, J. C. (1996). Foundations for Programming Languages. MIT Press (cit. on p. 20).

References

Pierce, B. C. (2002). Types and Programming Languages. MIT Press (cit. on pp. 28–35).

Pitt, D., Abramsky, S., Poigné, A. and Rydeheard, D., eds. (1986). Category Theory and Computer Programming. Vol. 240. Lecture Notes in Computer Science. Springer-Verlag. DOI: 10.1007/3-540-17162-2 (cit. on pp. 13, 14).

Plotkin, G. (1992). Post-graduate Lecture Notes in Advance Domain Theory (Incorporating the "Pisa Notes"). Electronic edition prepared by Yugo Kashiwagi and Hidetaka Kondoh. URL: http://homepages.inf.ed.ac.uk/gdp/ (visited on 29/07/2014) (cit. on p. 20).

Poigné, A. (1992). Basic Category Theory. In: Handbook of Logic in Computer Science. Volume 1. Ed. by Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. Clarendon Press, pp. 413–640 (cit. on pp. 13, 14).

Ramos, E. (2022). The Graph Minor Theorem Meets Algebra. Notices of the AMS 69.8, pp. 1297–1305. DOI: 10.1090/noti2522 (cit. on p. 53).

Russell, B. [1903] (1938). The Principles of Mathematics. 2nd ed. W. W. Norton & Company, Inc (cit. on pp. 25–27).

Scott, D. S. (1980). Relating Theories of the Lambda Calculus. In: To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism. Academic Press (cit. on pp. 11, 13, 14).

References

Shustek, L. (2022). An Interview with Dana Scott. Communications of the ACM 45.8, pp. 25–29. DOI: 10.1145/3544551 (cit. on p. 19).

Wadler, P. (2015). Propositions as Types. Communications of the ACM 58.12, pp. 75–84. DOI: 10.1145/2699407 (cit. on pp. 40–43).

Yang, E. Z. (2010). ezyang's blog. Hussling Haskell Types into Hasse Diagrams. URL: http://blog.ezyang.com/2010/12/hussling-haskell-types-into-hasse-diagrams/ (cit. on p. 50).