

1 Introduction

1.1 The Social Life of Objects

Questions that we are frequently asked are: What is category theory good for? Why should I use category theory? These questions usually indicate a genuine and healthy reaction to the proposal of a new piece of mathematics that one is invited to learn, similar to the reaction we have each time we are asked to change our eating habits and, say, replace butter with olive oil when cooking... How is this going to make us happier? Or healthier? When is the change justified?

The question also indicates that the nature and role of category theory is not completely clear to many people. The way we like to present category theory is as a toolbox similar to set theory: as a kind of mathematical lingua franca in the sense that it can be used for formalising concepts that arise in our day-to-day activity. It constitutes, however, a richer toolbox in the sense that the instruments that it provides are more sophisticated and thus make it easier to model situations that are more complex and that involve structured objects. On the other hand, because these instruments are more sophisticated than those of set theory, they require a dedicated learning effort. Briefly, in category theory one can do as much as in set theory, in an easier way when it comes to formalising and relating different notions of “structure”, but at the cost of learning a few more concepts and techniques.

So, I would like to reformulate the original question as: Having been brought up to think about the world in terms of sets, why should I now change and use another frame of reference? The purpose of this book is to convince you that this change of reference is worth making, that is, that this book will have been worth reading and that the concepts and techniques that it introduces belong in the mathematical toolbox of the software engineer.

For many of “us”, the key factor that justifies this change is related to the fact that, whereas concepts in set theory are typically formalised extensionally, in the sense that a set is defined by its elements, category theory provides a more implicit way of characterising objects. It does so in terms of the relationships that each object exhibits to the other objects in the universe of discourse. The best summary of the essence of category theory

that I know is from the logician Jean-Yves Girard.¹ For him, category theory characterises objects in terms of their “social lives”.

In my opinion, this focus on “social” aspects of object lives is exactly the reason for the applicability of category theory to computing in general, and software engineering in particular. To realise why this is so, one just needs to think that current software development methods, namely object-oriented ones, typically model the universe as a *society* of interacting objects. Agent-oriented methods are based on the same societal metaphor. This focus on interaction is not accidental; it is an attempt at tackling the increasing complexity of modern software systems. In this context, complexity does not necessarily arise from the computational or algorithmic nature of systems, but results from the fact that their behaviour can only be explained as emerging from the interconnections that are established between their components. By promoting interactions as a focal point in the definition of the structure of a system, one also brings software to the realm of natural, physical and social systems, something that seems to be essential for the development of well-integrated systems. Category theory is advocated as a good mathematical structure for this integration precisely because it focuses on relationships and interactions! The work of Goguen on general systems theory [52, 53, 64], and recent books like [94] show exactly that.

This is why many of the examples that are used throughout this book address what has become known in software engineering as software architecture [50], i.e. precisely the study of the gross modularisation principles that should allow us to design systems as possibly standard structures of smaller components and the interconnections between them. The focus that category theory puts on morphisms as structure-preserving mappings is paramount for software architectures because it is the morphisms that determine the nature of the interconnections that can be established between objects (system components). Hence, the choice of a particular category can be seen to reflect, in some sense, the choice of a particular “architectural style”. Moreover, category theory provides techniques for manipulating and reasoning about system configurations represented as diagrams. As a consequence, it becomes possible to establish hierarchies of system complexity, allowing systems to be used as components of even more complex systems (i.e. to use diagrams as objects), and for inferring properties of systems from their configurations.

The ultimate conclusion that we would like the reader to draw is that high-school education could well evolve in a way that equips our future generations with tools that are more adequate for the kind of systems that they are likely to have to develop and interact with. We believe that the

¹ What stronger evidence would one need of the close relationship between logic and category theory...

teaching of mathematics could progressively shift from the set-theoretical approach that made it “modern” some decades ago to one that is centred on interactions. This is, of course, a big challenge, mainly because it is not enough for the mathematical theory to be there; the right way needs to be found for it to be transmitted. We believe that recent books such as [75] are putting us on a path towards meeting this challenge, and we hope this book makes a further contribution. However, we will be satisfied if the rationale for such a shift can somehow emerge from the way we motivate and present the basics of category theory.

1.2 Categories Versus Sets

Let us consider an example in order to make clear the difference in approaches between set theory and category theory. For this purpose, there is nothing better than showing how certain mundane set-theoretic constructions are modelled in category theory.

Consider, for instance, the characterisation of the empty set. In set theory, the empty set is characterised precisely by the property of not having any elements. There are several ways in which we can say this using a formal notation. Here is one:

$$(\forall x)x \notin \emptyset.$$

The important point here is that any characterisation requires the use of the membership relation \in , i.e. the characterisation is made with respect to the elements that belong to the set.

Consider now the characterisation of the empty set in category theory. As discussed above, such a characterisation involves the definition of a “social life” of sets, i.e. it has to be made relative to the way sets interact with one another. Hence, the obvious question to ask first is “What is the social life of sets”?

The first important thing to understand is that category theory does not provide an answer to questions like this one; it is up to whoever is formalising a particular domain of discourse to come up with a definition of “social life” that is convenient. Convenience here has to be measured against the formalisation activity that is being undertaken, possibly as an abstraction of real-world phenomena. Hence, it is not subject to mathematical proof. Category theory requires some basic properties of such a “social life” so that the whole mathematical machinery that we are about to describe can be applied successfully. Such basic properties are defined in Sect. 2.2. In a nutshell, they prescribe ways in which one is related to oneself and the way one’s relations’ relations are our own relations.

Each time, during both classes and industry-oriented tutorials, our audiences were first confronted with the need for defining a social life for sets, the immediate answer was:

Set A is related to set B iff $A \subseteq B$.

Discussing the reason why this answer comes up spontaneously is well beyond the scope of this book.¹ The characterisation of the empty set based on this social life of sets is quite easy: the empty set is the only set that is related to every other set by this particular relationship.

Prompted for another definition of social life, the following answer was given several times:

Set A is related to set B iff $A \cap B \neq \emptyset$.

According to this definition, the empty set is the only set that is not related to any set; all non-empty sets relate at least to themselves. Incidentally, we shall see in Sect. 2.2 that this definition of social life does not define a category, one of the reasons being that, in a category, every object is at least related to itself in a canonical way. Nevertheless, the example is useful for showing that changing the definition of social life may lead to quite different characterisations of the same objects.

We typically have to force the audience to come up with the “standard” definition of social life between sets:

A social relationship between a set A and a set B is given as a total function $f: A \rightarrow B$.

The characterisation of the empty set in this case is very similar to the first one: the empty set is such that, given any other set A , there is one, and only one, function to A – the empty function. Notice that, although concepts such as “contains”, “intersection” and “function” are defined via the membership relation, the characterisation of the empty set given in the three cases does not involve it directly.

For further evidence that the nature of objects changes according to the “social life” that is of interest, consider the characterisation of singletons. According to the first definition, a singleton is such that only the empty set and the singleton itself relate to it. In the second case, a singleton is such that it relates to the sets that contain the element: $\{a\}$ relates to B iff $a \in B$. This is a good evidence for the fact that the second definition is not very “categorical”: it reduces to set membership and makes the identity of the elements visible. This is also the case with the first definition: in both cases, two singletons are socially equivalent, in the sense that they relate to any other set in the same way, iff they are equal. Incidentally, a possible set-theoretic characterisation of a singleton set $\{a\}$ is:

$$x \in \{a\} \text{ iff } x = a.$$

¹ Nevertheless, the author is willing to collaborate in any research project that aims at understanding the psychology of modern formalism.

The use of equality between elements makes clear the need to look inside the set.

The third definition characterises singletons as sets into which there is one, and only one, total function from any other set. Indeed, the function being total, a singleton offers no choice for the mapping to be established: everything is mapped to this single element. According to this definition, two singletons cannot be distinguished because they relate in exactly the same way with the other sets. This is quite intuitive because, not being able to use set membership, we cannot look inside the singletons and notice that they have different elements. Hence this social life is a better abstraction from set membership than the other two.

The “social” way of characterising objects is, in fact, similar to the way, in object-oriented programming, the view of objects that matters for building systems is that of the methods that objects make available to the other objects to interact with. However, the view that matters for their implementation may be different, reflecting the fact that different uses reflect different structural views of the same concept (which in category theory means different categories).

Other examples could be given from branches of engineering or our day-to-day praxis. For instance, another example is the way we understand devices as mundane as hi-fi stereo systems. When assembling a hi-fi system from separate components it is important to know how each component can be connected to the others; the way they are implemented in terms of microcircuits probably explains the restrictions on the connections that can be established, but it is something that a user would like to see abstracted away. The characterisation of human behaviour can also be used as an example. There is probably a neuro-physiological justification for what we call a “shy” or “expansive” person, but the meaning that we normally attach to such features of human nature is derived from the way people interact with us. The final word, however, comes from Saint Exupéry: “The meaning of things lies not in the things themselves, but in our attitude towards them”.

In brief, like all mathematics, category theory is about providing us with abstraction mechanisms. In our opinion, the fact that these mechanisms relate to interaction make category theory particularly suited for certain aspects of computing science, and software engineering in particular.

1.3 Overview of Typical Application Areas

In this section, we summarise some of the main applications areas of categorical techniques we know, with references to the literature, bearing in mind that our focus is software engineering and not computer science as a

whole. We strongly suggest [26, 57] for excellent introductions to (and overviews of) the field.

1.3.1 General Systems Theory

Goguen started exploring category theory as a mathematical toolbox in the early 1970s, applying it to general systems theory [52, 53, 64]. The famous motto [57]

“given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them”

was first applied to mathematical models of system behaviour. We also find in this area the origins of latter applications to object-oriented modelling [58]. In fact, one of Goguen’s early works is even called, very appropriately, “objects” [54].

But this is just one example of the many applications of category theory to the area of general systems. We would like to encourage the non-specialist to get acquainted with the field through one of the recent books that have appeared in the wider market of science, like [71, 94].

1.3.2 Algebraic Development Techniques

This is one of the most typical areas of the application of category theory to computer science. One of the earlier and most influential movements started in the second half of the 1970s with the work of a group of researchers at IBM (Joseph Goguen, Jim Thatcher, Eric Wagner and Jesse Wright), very aptly named ADJ (for *adjunction*), around the initial semantics of abstract data type specification [65]. The pioneering work of Rod Burstall and Joseph Goguen around the language CLEAR [17, 18] showed how simple universal constructions (e.g. colimits) could be used to give semantics to operations for structuring specifications (e.g. computing the sum of two specifications, parameterising a specification, etc.). See also [95, 96] for some more advanced examples and [101] for an example of tool support. The area produced textbooks such as [28, 29, 76, 92] as well as surveys [8, 26] that provide a good showcase for “categories at work”, namely in what concerns elegance and economy of means.

One of the finest hours of this research programme was the development of the theory of institutions, a categorical formalisation of the notion of logic developed by Goguen and Burstall in the early 1980s [62]. The aim of this effort was to provide a mathematical framework in which the specification building operators defined for the language CLEAR could be made independent of the underlying logic and thus available for other specification formalisms. The theory of institutions is, in our opinion, one of the best examples of the usefulness of category theory as a unifier of concepts and techniques developed by different research teams in response

to different or similar needs. As a science, computing is still very young and, hence, fragmented in that it is often difficult to find the “universal laws” that we recognise as the objects of study in other sciences. Through the theory of institutions, category theory was shown to be a powerful tool for abstracting from individual and uncoordinated efforts some universal laws (or “dogmas” as they are called in [57]) that apply to specifications in general [95]. This categorical framework was subsequently extended and put to use, for example, in applying algebraic specification techniques to other computational paradigms (e.g. object-oriented) and defining ways for specifications to be mapped from one formalism to another.

1.3.3 Concurrent and Object-Oriented Systems

Concurrency theory is another area to which categorical techniques have been applied in the “engineering” view that interests us in this book. This domain of application is dominated by the work of Glynn Winskel [106, 107, 108], who showed how models for concurrent system behaviour like transition systems, synchronisation trees and event structures can be formalised in category theory. The idea is that each process model is endowed with a notion of morphism that defines a category in which typical operations of process calculi are given as universal constructions. This work was continued in the 1990s in collaboration with Mogens Nielsen and Vladimiro Sassone [97], exploring the use of adjunctions as a means of translating between different models. The chapter [109] provides an excellent overview of this exciting application area.

This stream of work (including [15]) was used for giving semantics to object-oriented systems by Félix Costa, Hans-Dieter Ehrich, and Amílcar and Cristina Sernadas [20, 21, 25], also with contributions from Goguen [24]. In a nutshell, these authors showed how universal constructions can be used to express object-oriented features like encapsulation, inheritance and composition in concurrency models endowed with richer notions of state. The work of Goguen on sheaf-theoretic models [58] also establishes an interesting connection to the earlier work on general systems theory.

Applications of category theory to the (logical) specification of concurrent and object-oriented systems can also be found. These include our work in the late 1980s and early 1990s [38, 39, 40, 43]. Basically, we showed how the techniques developed for the modularisation of equational and first-order logic specifications of abstract data types could be applied to concurrent and object-oriented systems by using modal and temporal logics developed for the specification of reactive systems [13].

Briefly, the idea was to fit the specification of such systems in the “institutional” picture as set up by Goguen and Burstall [32, 33]. We later showed that our work is directly related to the General Systems tradition [31], in the sense that the module structure defined by our categorical formalisation is directly compositional over the run-time structure of the sys-

tem. This led to applications of the categorical techniques to parallel program design [44], in which the notion of morphism captures what in the literature is known as superposition or superimposition [19, 48, 72].

There is also a substantial amount of work in the application to concurrent and object-oriented systems of the algebraic approach to abstract data type specification [7], most notably by Egidio Astesiano and his colleagues in Genova. Basically, these approaches present different alternative ways of bringing states and transitions to the universe of discourse of abstract data types.

1.3.4 Software Architectures

Applications of categorical techniques to the semantics of interface description and module interconnection languages were developed by Goguen in the early 1980s [55] and more recently recast in the context of the emerging interest in software architecture [59]. These applications are in the tradition of the algebraic approach to abstract data types, more specifically in what has become known as “parameterised programming” [56], in the sense that they capture functional dependencies between the modules that need to be linked to constitute a given program.

The view of architectures that is captured in this way is somewhat different from the one followed in the work of Dewayne Perry, David Garlan and other researchers who have launched software architecture as we know it today [14, 50, 99]. This more recent trend focuses instead on the organisation of the *behaviour* of systems as compositions of components ruled by protocols for communication and synchronisation. As explained in [2], this kind of organisation is founded on *interaction* in the behavioural sense, which explains why formalisms like the chemical abstract machine (or CHAM) [16] are preferred to the functional flavour of equational logic for the specification of architectural components.

This why, in the early 1990s, we proposed a categorical toolset for architectural description based on our work on the formalisation of parallel program design techniques [34, 42]. The idea is that, contrarily to most other formalisations of architectural concepts that can be found in the literature, category theory is not another semantic domain in which to formalise the description of components and connectors. Instead, through its universal constructions, it provides the very semantics of *interconnection*, *configuration*, *instantiation* and *composition*, i.e. that which is related to the gross modularisation of complex systems.

1.3.5 Service-Oriented Software Development

However, there is more to category theory than the ability to support such interaction-based views of system behaviour. By relying on “local naming” (as shown by the impossibility of distinguishing between singleton

sets), category theory requires all such interactions to be modelled explicitly and outside the participating objects. This is one of the aspects that distinguishes service from object-oriented system development.

Clientship, i.e. the ability to establish client/supplier relations between objects, leads to systems of components that are too tightly coupled and rigid to support the levels of agility that are required to operate in environments that are “business time critical”, namely those that make use of Web-services, Business-to-Business (B2B), Peer-to-Peer (P2P), or otherwise operate in what is known as “internet-time”. Because interactions in object-oriented approaches are based on *identities*, in the sense that through clientship objects interact by invoking specific methods of specific objects (instances) to get something specific done, the resulting systems are too rigid to support such levels of agility. Any change in the collaborations that an object maintains with other objects needs to be performed at the level of the code that implements that object and, possibly, on the level of the objects with which the new collaborations are established. On the contrary, interactions in a service-oriented approach should be based only on an abstract description of what is required, thus decoupling “what one wants to be done” from “who does it and how”. This is precisely the discipline of interconnection that category theory enforces.

Our last claim, in support of the last paragraph of Sect. 1.1, is that category theory is definitely the mathematics of the Internet age (and beyond)!

1.4 What Can Be Found in This Book

This book emerged from tutorials and courses given in the past few years, both in academia and in more industry-oriented fora like OOPSLA, ObjectWorld, TOOLS, ECOOP and FME. I am particularly grateful to the University of Lisbon and the Technical University of Lisbon for the opportunities that they gave me to lecture on much of the material covered in the first two parts, at both the undergraduate and postgraduate levels. These parts of the book were also covered in lectures given at the University of Coimbra, the Institute for Languages and Administration in Lisbon (ISLA) and the Federal University of Rio Grande do Sul (Brazil). Part III was used in a 20-hour course on CommUnity and software architecture that I gave as part of a postgraduate programme of the University of Pisa, as well as at the Summer School on Generic Programming (Oxford 2002). All this experience showed that there was an audience for this particular way of teaching category theory.

The book is structured in three parts, leaving room for different reading/teaching paths to be followed. With respect to most other books in the market, this one uses examples of a different nature, focusing on and giv-

ing more emphasis to aspects that are less common in other fields of computing. It also adopts a different pace altogether.

- Part I, i.e. Chaps. 2–5, covers some of the basics of category theory, including the “bare essentials” that are addressed in any book, from graphs to universal constructions and functors. However, a different emphasis and tone are used that are meant to be more appealing and accessible to software engineers. It is hoped that mathematically mature readers may appreciate a different way of exposing and illustrating these familiar concepts and constructions.
- The material included in Part II, Chaps. 6 and 7, is of a more advanced nature, not only because it is more challenging from a mathematical point of view but also because it makes appeal to an additional level of maturity in so far as computing science is concerned, namely the use of multiple formalisms as supporting complementary viewpoints. There, the reader will find material that only a few other books cover in comparable depth, with a strong emphasis on functor-based constructions like fibrations, and ending with a covering of adjunctions that deviates somewhat from the standard coverage. Again, examples are drawn from areas that have normally been confined to papers such as institutions and models of concurrency. The sections in this part should be accessible to anyone with basic knowledge of category theory, but a quick travel through Part I will help the reader become familiar with the notation and the examples that are used in Part II.
- Part III offers the chance of seeing category theory at work in a more ambitious project – giving semantics to CommUnity, a prototype language for architectural modelling. This part can be ignored by readers who are not particularly interested in the applications to software engineering. On the other hand, it can be followed, to a large extent, without the material exposed in Part II. This means that a novice to category theory but who is interested in software engineering, or anyone whose goal is to understand CommUnity or set up a course that aims at teaching (any subset of) the material in Part III, can safely skip Part II. It may be necessary to go back to Part II in order to understand in full the mathematical structures that relate to software architecture, but this may be done once the reader feels more at ease with the mathematics.

Parts I and II use examples taken directly from Meyer’s book on Eiffel [86] to illustrate definitions and constructions. It is hoped that readers who are not familiar with the particular notation of Eiffel, but are used to object-oriented modelling, will be able to understand the examples without much effort. The choice for Eiffel instead of a more modern language has to do with two facts. On the one hand, it *was* modern when the book started to be written.... On the other hand, it is one of the object-oriented

languages that has solid foundations, which allows it to be used to illustrate many of the mathematical constructions that we cover in the book.

The use of Eiffel allows us to illustrate applications of category theory to the more static aspects of system modelling, namely to what is related with classification (inheritance). Other examples are brought to bear from specification theory that involve logics and algebraic models for system behaviour in order to show how the more dynamic and evolutionary aspects can also be handled. These aspects come together in Part III.

This means that different reading/teaching paths can be established that are based on the examples: one can follow the “Eiffel path” for a lighter approach, perhaps more suitable for “practitioners”; or one can follow the “specification theory path” that will enable the “scientists” to visit more challenging places and reach the end of Part II. But who are the true software engineers if not the people who can combine both? Therefore, the best approach is to read the book from the beginning to the very end!



<http://www.springer.com/978-3-540-20909-6>

Categories for Software Engineering

Fiadeiro, J.L.

2005, XIV, 250 p., Hardcover

ISBN: 978-3-540-20909-6