A comprehensive application of category theory to semantics of modelling language

Carlos Ramisch, Iulia-Maria Hudita

ENSIMAG, INP Grenoble  Math for Fun  January 21., 2008
{ramischc, huditai}@ensimag.fr

Abstract: Since their creation, category theory has developed and is nowadays considered a powerful tool in mathematics and in computer science. This paper is organized in two parts: an introduction to categories and a complete application of them to a software engineering context. In the first part, we will justify why categories are so widely used and where can we find them in the real world. Then, we will introduce in a very pedagogic way some basic concepts, including categories themselves. Afterwards, we will introduce some other important notions like diagrams and sketches, that will be used in the second part of the paper to describe an example based on (Diskin, Wolter, 2007). This example defines the semantics of a software modelling language using categorical tools. Finally, we will present a personal point of view on the subject and explain why do we have fun with category theory.

1. Introduction

The philosopher's stone of mathematicians is the utopian generic theory that could explain all other theories. Unfortunately, it has already been proved that this is impossible. The incompleteness theorem proved by Kurt Gödel in 1931 [2] confirms that there is no such super-theory thus if it existed it would not be able to explain itself.

However, in 1945 the North-American mathematicians Saunders Mac Lane and Samuel Eilenberg established an elegant theory that could fill the requirements in order to be a best approximation of the philosopher's stone, general enough to explain other fields of mathematics. This theory, which they called category theory, was developed through the second half of the 20th century and is nowadays a very powerful tool in mathematics and especially in computer science.

Category theory has a great power of expression because it is based on the composition of very natural and therefore comprehensible constructions. Calculations and deductions are made through the use of graphs and diagrams, which are structures to which computer scientists are used to. Definitions in category theory are quantified logic statements over commutative of diagrams, and this makes them easy to understand. Therefore, it is a surprise that even if not based on complex concepts, category theory can also be so powerful.

The power of category theory resides in the idea of composition. The definition of a category
itself contains composition. Categories and functors are to be composed in order to generate more abstract structures. With both ideas, diagrammatic theorems and compositional thinking, we can found categories in a large range of applications.

2. Bibliographic review

Categories are used, for example, to explain the formal semantics of programming languages. The semantics of concurrent constructions in programming languages can be expressed easier with categorical approaches than with Calculus of Concurrent Systems or other similar semantics description languages. Even traditional formal semantics, like operational, denotational and axiomatic, can be simplified and even replaced by semantics based on categorical logic (Hildebrandt, 2002).

Also non-trivial applications can be found, like the use of categorical composition to render the animation of scenes in a film. We can define the fragments of the animation together with the functors of the interactions between these fragments. Applying a categorical product engine, the whole animation will be rendered without the need of storing and calculating every frame. This can be particularly useful when a scene involves a great number of characters, like an army for example, where the characters show a periodic behaviour (Grandi & Menezes, 2003).

We can also find programming languages that are completely categorical. The language Nautilus, proposed by (Menezes et. al., 2006), is a language with non-trivial constructions which is object-oriented and naturally concurrent. With this language, complicated synchronization mechanisms, like semaphores and monitors, are replaced by automata-based operators with clear semantics.

Hagino, in (Hagino, 1987) also proposes a functional programming language. He focuses his work on datatypes, replacing traditional formalizations based on FOL by a more abstract formalization that uses category theory. In his work, Hagino uses constructions like products, co-products, and adjunctions to create what he calls $F,G$-dialgebras, used to define the types in his programming language. This can prove that his programming language is type-safe and loop-safe.

Finally, (Nelson & Rossiter, 1995) provide a formal model for object-oriented databases. Their work deals with some major problems in these kind of databases like queries and views. They have also an elegant model for concepts like keys, relationships and aggregations.

3. Basic category theory

In order to understand the rest of this paper, we will start with a short introduction to the ideas on the very base of category theory, according to (Barr & Wells, 1990). First of all, we need to define a graph from a categorical point of view. The definition 1 is a textual description of the presented diagram. In category theory, most part of the constructions can be expressed by diagrams and eventual constraints over them.
Definition 1  (Directed) Graph

A directed graph or simply a graph is a 4-uple $G = < g_0, g_1, \delta_0, \delta_1 >$ where $g_0$ is a set of nodes or objects, $g_1$ is a set of arcs or morphisms and the functions $\delta_0, \delta_1 : g_1 \to g_0$ associate each arc of $g_1$ with a node of $g_0$. These functions are also called source and target, respectively.

In graph theory, it is common to define source and target as a single function that takes each arc to an ordered pair of nodes, representing the source and the target of that arc. We preferred to define two functions $\delta_0$ and $\delta_1$ because that makes the definition of a graph more general and closer to the definition of a category. Nonetheless, we will use the simplified notation $f : n_0 \to n_1$, where $f$ is an arc and $n_1$ and $n_2$ are nodes, meaning that $\delta_0(f) = n_0$ and $\delta_1(f) = n_1$. In the example below, we see the definition of a graph by the enumeration of all its elements. Normally, we don't write the whole definition, but we represent the objects and the arrows connecting them according to the source and target functions.

Example 1  A graph and its corresponding diagrammatic representation

$G = < g_0, g_1, \delta_0, \delta_1 >$

$g_0 = \{ a, b, c \}$
$g_1 = \{ 1, 2, 3 \}$
$\delta_0 = \{ a \rightarrow 1, b \rightarrow 2, c \rightarrow 2 \}$
$\delta_1 = \{ a \rightarrow 2, b \rightarrow 2, c \rightarrow 3 \}$

Definition 2  Reflexive graph

$Gr$ is a reflexive graph if and only if $Gr$ is a graph with an associated function $\iota : g_0 \to g_1$ (called identity function) that respects the following constraint:

$\forall n \in g_0, \delta_0(\iota(n)) = \delta_1(\iota(n)) = n$

It is usual to write $i_n$ for the identity morphism associated to the object $n$ by the function $\iota$. We show the free reflexive graph generated from the graph in example 1 adding the identity morphisms to each node.
Example 2  A reflexive graph and its corresponding diagrammatic representation

\[ \text{Gr} = < \text{g}_0, \text{g}_1, \delta_0, \delta_1, \iota > \]

\( \text{g}_0 = \{ a, b, c, i_1, i_2, i_3 \} \)

\( \text{g}_1 = \{ 1, 2, 3 \} \)

\( \delta_0 = \{ a \to 1, b \to 2, c \to 2, i_1 \to 1, i_2 \to 2, i_3 \to 3 \} \)

\( \delta_1 = \{ a \to 2, b \to 2, c \to 3, i_1 \to 1, i_2 \to 2, i_3 \to 3 \} \)

\( \iota = \{ 1 \to i_1, 2 \to i_2, 3 \to i_3 \} \)

\[ \begin{array}{c}
    i_1 \\
    1 \\
    \text{a} \\
    2 \\
    b \\
    i_2 \\
    i_3 \\
    3 \\
\end{array} \]

Moving from a reflexive graph to a category can be easily done by introducing the notion of compositionality. A category is nothing more than a graph with an identity and a composition function. Its formal definition is:

**Definition 3  Category**

A category \( C \) is a reflexive graph with an associated function \( \circ : \text{g}_1 \times \text{g}_1 \to \text{g}_1 \) (called composition function) that respects the following two constraints:

i) Identity: \( \forall a : n_1 \to n_2 \in \text{g}_1, \iota(n_1) \circ f = f \circ \iota(n_2) = f \)

ii) Associativity: \( \forall a_1 : n_1 \to n_2, a_2 : n_2 \to n_3, a_3 : n_3 \to n_4 \in \text{g}_1, a_1 \circ (a_2 \circ a_3) = (a_1 \circ a_2) \circ a_3 \)

Example 3  A category and its corresponding diagrammatic representation

\[ \text{C} = < \text{g}_0, \text{g}_1, \delta_0, \delta_1, \iota, \circ > \]

\( \text{g}_1 = \{ 1, 2, 3 \} \)

\( \text{g}_2 = \{ a, b, c, i_1, i_2, i_3 \} \)

\( \delta_0 = \{ a \to 1, b \to 1, c \to 2, i_1 \to 1, i_2 \to 2, i_3 \to 3 \} \)

\( \delta_1 = \{ a \to 2, b \to 3, c \to 3, i_1 \to 1, i_2 \to 2, i_3 \to 3 \} \)

\( \iota = \{ 1 \to i_1, 2 \to i_2, 3 \to i_3 \} \)

\( \circ = \{ (i_1, i_1) \to i_1, (i_2, i_2) \to i_2, (i_3, i_3) \to i_3, (i_1, a) \to a, (a, i_2) \to a, (i_1, b) \to b, (b, i_3) \to b, (i_2, c) \to c, (c, i_3) \to c, (a, c) \to b \} \)

\[ \begin{array}{c}
    i_1 \\
    1 \\
    \text{a} \\
    2 \\
    b \\
    i_2 \\
    i_3 \\
    3 \\
\end{array} \]
Normally, we will suppose that functions \( \iota \) and \( \circ \) are implicitly defined by constraints (i) and (ii) and we will not enumerate their elements like in the preceding examples. Actually, the diagram of a category looks like the diagram of a reflexive graph that respects the imposed constraints. That is a direct consequence of the formal definition of a diagram. Until now, we used the term diagram indiscriminately to name a graphical representation of a complex structure, without regarding to the following formal definition. We will here define a diagram as a graph homomorphism.

**Definition 4  Graph homomorphism**

A graph homomorphism \( F : G \to H \) for the graphs \( F \) and \( G \) is a pair of functions \( f_0 : g_0 \to h_0 \) and \( f_1 : g_1 \to h_1 \) that conserves the internal structure of the graph. This conservative property can be expressed by:

\[
\forall a : n_1 \to n_2 \in g_1, \delta_{0H}(f_1(a)) = f_0(n_1) \land \delta_{1G}(f_1(a)) = f_0(n_2)
\]

or else by the diagram

\[
\begin{array}{ccc}
g_1 & \xleftarrow{\delta_{1G}} & g_0 & \xleftarrow{\delta_{2G}} & g_1 \\
g_1 & \xrightarrow{f_1} & h_1 & \xleftarrow{\delta_{1H}} & h_0 & \xleftarrow{\delta_{0H}} & h_1 \\
\end{array}
\]  

(1)

**Definition 5  Diagram**

A diagram over a graph \( G \) of shape \( I \) is a graph homomorphism \( D : I \to G \). \( I \) is called the shape graph of \( G \).

There are two consequences from this definition: 1) A diagram may have two or more objects or morphisms with the same label, and they will represent the same object and morphism in the graph \( G \) (as in (1), where \( g_1, h_1 \) and \( f_1 \) appear twice, not meaning that a graph has two distinct sets of morphisms); and 2) two diagrams with the same shape graph \( I \) but with different mappings \( D \) are actually two different diagrams. Obviously, the concept of diagram can be used with all the structures that are based on a graph, including a category.

Finally, we will introduce the concept of sketches in the next section. This is a non-trivial concept that will be used throughout the rest of this work.

### 4. Sketches

In mathematics, there is always a need for specifying structures. In logic, this is done through signatures, models and specifications. Computer scientists are used to build specifications up from formal grammars and semantic rules. In category theory, sketches play the role of specifications. The most general type of sketches are linear sketches.

**Definition 6  Linear sketch**

A linear sketch \( S \) is formed by a graph \( G \) and a set of diagrams \( D \) over \( G \). The morphisms of the graph \( G \) are often called operations of \( S \).
Sketches are key tools in the application that we will present in the second part of this paper. In the following two sections, we will resume some aspects of the work of Diskin and Wolter in (Diskin, Wolter, 2007), who apply the concepts introduced in the first part of the paper to a software engineering context. Section 5 discusses the importance of a clear semantics for graphical modelling languages such as UML [4], nowadays largely used in software analysis and conception. Section 6 shows a way of applying categorical concepts in order to get a non-ambiguous semantics for models, according the proposition of Diskin and Wolter.

5. Semantics of modelling language

In the beginning of the referenced article we are introduced with the notion of *modeling language* (diagrams or diagrammatic models). Even if usually these diagrams are specified from the syntax point of view, the semantic aspect remains undefined.

Because of the invasion of the model-centric trends in the software industry, the semantic definitions become imperative. The main goal is generating code directly from well defined models and it can be called Model-Driven Engineering (MDE), Model-Driven Development (MDD) or Model-Driven Architecture (MDA).

For this Model-Driven approaches it is essential to have a precise formal semantics for diagrammatic notations. The majority of the building formal semantics are founded on the first-order (FO) or other logical systems derived from string-based formulas. There is a mismatch between the logical machineries used for formalization and the internal logics of the domains intended to be formalized.

Diskin and Wolter consider that as the modeling, meta-modeling and modeling language design are more and more present in the industry, it is very important for a software engineer to be familiarized with it. Throughout the article several aspects of the FOL and therefore certain aspects of category theory are introduced.

6. Categorical semantics for modelling diagrams

A labeled diagram is a graph-based analog of a formula. A logic formula $P(x_1 \& x_n)$ is just a specific syntax for a labeled diagram whose shape is a set and not a graph. Therefore, a sketch is a set of graphs: atomic formulas over a fixed context of names (variables).

The notion of *multi-sketches* and the *dependencies* (arrows between predicate symbols) are very important. Because of dependencies, a signature of predicate symbols becomes a graph, even a category, which makes the entire machinery much easier to manipulate.

We need to understand why categorical sketches are relevant. A very important characteristic of the Model approaches in software engineering is the *two-dimensionality* (2D), that allows representing the structure by a graph, for example:

- entities and relationships;
- objects and links;
- states and transitions;
- events and messages;
- agents and interactions.
A system can be represented by multiple graphs interrelated between themselves. At this point, we can talk about a graph-based structure on the metalevel (the nodes represent graphs that model different aspects of the system and the arrows the different relations and interactions between them).

A graph-based approach to specifying structures, that is oriented towards the relationships between domains more than their internal contents, can be found in category theory. Charles Ehresmann in the 60s introduced the concept of sketch for category theory. Sketches are defined in section 4.

In the reference article the authors give a sample of sketching a diagrammatic notation in order to prove the essence of the classical sketch approach to specifying data. In the first part we can see a simple ER-diagram that describes a configuration of sets and mappings presented in the second part.

Another definition that will be useful is the extension of the notion of sketches to generalized sketches. In the example given by Diskin and Wolter, in order to declare the fact that en entity is a binary relation, they were forced to introduce some auxiliary elements into the specification. The extension of a compound node is not stored and some elements from the original diagrams cannot be seen directly in the sketch, but it can be deduced by derivation.

Therefore, before being able to establish a semantic meaning to ER-Diagrams’ elements and applying the SAAM Principle (Semantics-as-a-Morphism), non-trivial transformations need to be applied to the diagram. These transformations may seem artificial, misleading and unnecessary for a software engineer.

In the last part of the example, it is proven that the deficiency mentioned above of the classical sketch framework can be fixed without giving up the SAAM Principle. The key that demonstrates this idea is well-known in category theory: for a given span of mappings, its head is isomorphic to a relation if the legs (projection mappings) possess a special property of being jointly injective or jointly.

The idea of generalized sketches will be made clearer with the example of section 7, took from the reference paper.

7. An example

The example that we will show here talks about painting objects. One of the typical situations encountered in software modeling is categorization of objects into classes or types. There are:

- $O$: a set of objects
- $T$: a set of types
- $\tau : O \rightarrow T$ a typing mapping.

If $T = \{\text{red}, \text{blue}, \text{black}, \text{white}\}$, then $\tau$ would classify $O$ into sets of instances red, blue, black and white. In this case, $T$ is called the model and any mapping $\tau$ is a legal instance of the model.

If some extra rules are imposed, like the number of red objects must be always less than the number of blue objects (rule 1) or the number of black objects is less than the number of white objects (rule 2), the authors introduce a binary predicate $P$ into the specification language:

- rule 1 is defined by $P(\text{red}, \text{blue})$
- rule 2 is defined by $P(\text{black}, \text{white})$
The semantics of the predicate symbol $P$ is the set $[P]$ of $\tau$'s satisfying the requirement. In the predicate declaration $P(\text{red, blue})$, the bracketed part denotes the mapping:

$$D : \{1,2\} \rightarrow T = \{\text{red, blue, black, white}\}$$

where $d(1) = \text{red}$, $d(2) = \text{blue}$.

An instance $\tau : O \rightarrow T$ satisfies the declaration $P(d)$ if its inverse image along $d$ is an element of $[P]$.

Other predicates may be needed for this example, like $Q(1, 2, 3)$ such that a mapping $O \rightarrow \{1,2,3\}$ satisfies $Q$ if

$$|\tau^{-1}(1)| + |\tau^{-1}(2)| \leq |\tau^{-1}(3)|$$

If we add to our model the declaration $Q(\text{red, blue, black})$, all model's instances $\tau$ where the red and blue objects are more numerous than the black objects will be illegal.

Because the rule $Q (1, 2, 3)$ can be enforced only in the conditions $|\tau^{-1}(1)| > 10$ and $|\tau^{-1}(2)| > 10$. Therefore, an unary predicate $U$ with semantics $|\tau^{-1}(1)| > 10$ is introduced.

The signature is a graph $\pi$ of predicates (nodes) and dependency symbols (arrows), endowed with an arity graph mapping $\alpha : \pi \rightarrow \text{Set}$. At the same time, the models are sets of type $T$ and the instances are mappings between these sets. This fact allows the authors to say that their logic is based on the category of sets, $\text{Set}$.

### 8. Definitions

In this section we focus on familiarization with the syntactical side of the generic logic. Two notions are introduced: graph-based logic and string-based logic, which will both be referred to as Base.

Several important definitions are given and we will try to resume the most important of them:

A signature over a Base is a graph morphism $\alpha : \Pi \rightarrow \text{Base}$, where $\Pi$ is a graph of predicate and dependency symbols.

For an object $P$ in $\Pi$, $\alpha(P)$ is the arity of $P$ (noted $\alpha P$).

For an arrow $r : Q \rightarrow P$ in $\Pi$, $\alpha(r)$ is an arity substitution (noted $r^\alpha$).

A labeled diagram/formula over $G$ is a pair $(P, d)$, where $P$ is a predicate symbol and $d : \alpha(P) \rightarrow G$ is a morphism.

An ordinary sketch in the signature $\Pi$ is a pair $S = (G, \Phi)$ where $G$ is a base graph and $\Phi \subseteq F_m(G)$ ($F_m(G)$ denotes a set of formulas/diagrams over $G$). This definition is an extension of precedent definition 6, where we said that $\Phi$ is an arbitrary set of diagrams.

If we have $S = (G, \Phi)$ and $S_i = (G_i, \Phi_i)$ two ordinary sketches, their morphism is a substitution $s : G \rightarrow G_i$ such that $s^*(\Phi) \subseteq \Phi_i$, where $s^* : F_m(G) \rightarrow F_m(G_i)$.

**Theorem 1** Equivalence of Categories

For any signature $\alpha : \Pi \rightarrow [B \rightarrow \text{Set}]$ with $B$ a small category $\text{MSk}(\alpha)$ of multi-sketches is isomorphic to the functor category $[B^* \alpha \rightarrow \text{Set}]$. 
Theorem 2  Flattening

For any presheaf category $\text{Base} = [B \to \text{Set}]$ and signatures $\alpha_1: \prod_1 \to \text{Base}$, $\alpha_3: \prod_2 \to \text{MSk}(\alpha_3)$ there is a small category $\prod$ and a signature $\alpha: \prod \to \text{Base}$ such that

$\text{MSk}(\alpha_3) \cong \text{MSk}(\alpha)$

9. Discussion

First of all, we would like to remark an issue concerning the syntax of UML. In the reference paper, the authors say that "syntax of diagrams is specified (in the so called metamodel)". Nonetheless, this is not at all a didactic way of defining it, once one need already to know the syntax in order to understand it. We think that perhaps a categorical approach to syntax definition of models could solve this problems. Taking as basic tools graph grammars and categorical grammars, we could build a new way of deriving diagrams from specifications. Functors between the syntax of semantics would specify the relations between them.

The notion of sketches to define the specification of a theory and models of sketches to relate the specification with the instances is a very powerful tool. However, they are not an intuitive concept and are often not treated in basic category theory courses. Therefore, the comprehension of the reference paper is compromised, even if the main ideas are absorbed. We realized that there is a need of comprehensive and simple examples in all works of the area.

Categories require a highly abstract thinking. Some mathematicians call category theory abstract nonsense or general abstract nonsense [1]. This shows that there is also a funny point of view over category theory, once we can reach so high levels of abstraction that we start to doubt if it makes any sense at all. This is also a reason why categorical machinery must be implemented in functional programming languages in order to be applicable to real life situations like the one described here.

10. Conclusion

Graphical models are not a technological innovation in computer science. Since the very beginning, people used diagrams to model software artefacts, like flowcharts, Entity-Relationship models, state machines and so on. Although the Unified Modelling Language was created as a language that allows communication between programmers and system designers, it has been on the origin of the so called Model Based technologies. The idea of these approaches is that once we have a powerful diagrammatic language with a non-ambiguous semantic, there will be no need for writing code. Instead, we will write models that will automatically be converted into source-code.

There is a serious problem that concerns semantics of models and category theory is a very elegant way of solving these problems. With categories, we can make models based on generalized sketches that are well defined and very powerful mathematical structures. We presented here the basis for understanding why can this be done, and what are the tools needed to implement it.

Finally, we expressed a personal point of view on the bibliographic revision and the state-of-art presented in this paper. We also remarked that mathematics and especially category theory are not only powerful and useful, but can also amuse ourselves.
References


