

# 2 INTRODUCING CATEGORIES

## 2.1 Graphs

A distinctive attribute of Category Theory as a mathematical formalism is that it is essentially graphical. This means that most concepts and properties can be defined, proved and/or reasoned about using diagrams of a formal nature. This diagrammatic nature of Category Theory is one aspect that makes it so applicable to Software Engineering.

Therefore, it is not surprising that the most basic and, hence, the first definition in this book is that of graphs.

### 2.1.1 DEFINITION – graph

A *graph* is a tuple  $\langle G_0, G_1, src, trg \rangle$  where:

- $G_0$  is a collection<sup>5</sup> (of nodes),
- $G_1$  is a collection (of arrows),
- $src$  maps each arrow to a node (the source of the node),
- $trg$  maps each arrow to a node (the target of the node)

We usually write  $f:x \square y$  to indicate that  $src(f)=x$  and  $trg(f)=y$ .

Between two nodes there may exist no arrows, just one in either direction, or several arrows, possibly in both directions.

The attentive reader will have noticed that our very first definition still uses Set Theory, even if only informally. This may appear confusing, especially after our lengthy discussion of the merits of Category Theory with respect to Set Theory. However, we need a meta language for talking about graphs (and categories, and ...) which cannot, of course, be the object language itself (i.e. that of Category Theory). Hence, we will use the "informal" language that is typical of Mathematics which, as also acknowledged in the introduction, is full of set-theoretic concepts.

---

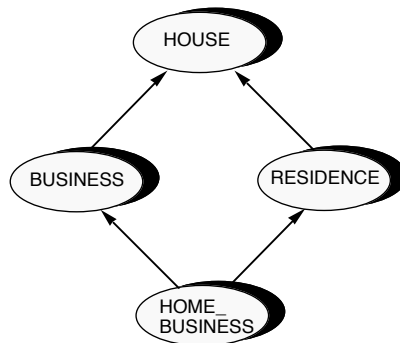
<sup>5</sup> Questions of “size” arise here because we shall soon be talking about the graph of graphs and constructions of a similar nature. This is why we use the term “collection” instead of “set”. See, for instance, [79] for a full treatment of such questions.

**2.1.2 EXAMPLE – sets and functions**

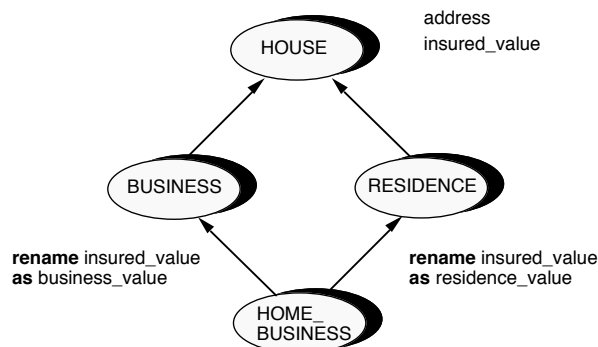
1. The most “popular” graph in the book (and in any other book we know) is the graph whose nodes are the sets and whose arrows are the total functions.
2. To illustrate the fact that different graphs may share the same nodes, we introduce what is, perhaps, the second most popular graph in the book – the graph whose nodes are, again, the sets but whose arrows are the partial functions, i.e. functions that may be undefined on given elements of the source set.

**2.1.3 EXAMPLE – class inheritance hierarchies**

A typical example of the use of graphs in Computing is class inheritance hierarchies. These are graphs whose nodes are object classes and for which the existence of an arrow between two nodes (classes) means that the source class inherits from the target class.

**2.1.4 A class inheritance hierarchy**

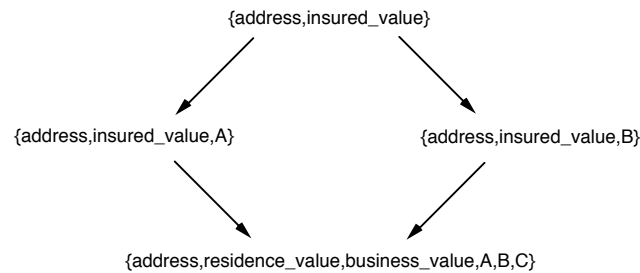
In class inheritance hierarchies, there exists at most one arrow between two nodes: either a class inherits from another or it does not. However, arrows can carry more information. For instance, when one class inherits from another one, some renaming of the features of the original class may be required. Such renamings may be associated with the arrows of a class inheritance diagram.

**2.1.5 A class inheritance hierarchy enriched with renamings**

Each such enriched class inheritance diagram defines a subgraph of the graph of sets and (total) functions – classes are represented through their sets of features and the

## 2.1 Graphs

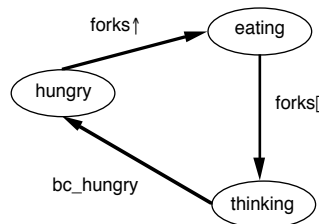
renamings through the functions that they induce. Notice that the arrows of the class inheritance hierarchy and the functions that operate the renamings point in opposite directions.



**2.1.6**      **The subgraph of sets and functions generated from the enriched class inheritance hierarchy. In this graph, *A* denotes a feature specific of *BUSINESS*, *B* denotes a feature specific of *RESIDENCE*, and *C* denotes a feature specific of *HOME\_BUSINESS***

### 2.1.7      **EXAMPLE – transition systems**

Another very common example of graphs is transition systems. Every transition system constitutes a graph whose nodes are the states and whose arrows are the transitions.

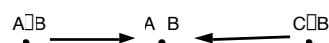


**2.1.8**      **A state transition system modelling the behaviour of a dining philosopher**

As hinted in the introduction, there are many (and deep) relationships between Logic and Category Theory. The next two examples convey some of these relationships. See [73] for a textbook on this subject.

### 2.1.9      **EXAMPLE**

One of the possible views that one can have of a "logic" is through the notion of a sentence being a consequence of, or derivable from, another sentence. This notion of consequence can be represented by a graph whose nodes are sentences and whose arrows correspond to "logical entailment".



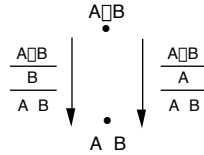
**2.1.10**      **An example of three nodes and two arrows of the consequence graph of propositional logic**

## 2. INTRODUCING CATEGORIES

We can add detail to consequence systems and distinguish between different possible ways in which a sentence can be a consequence of another, i.e. by associating proofs with arrows:

### 2.1.11 EXAMPLE

Every proof system constitutes a graph whose nodes are sentences and whose arrows are proofs.

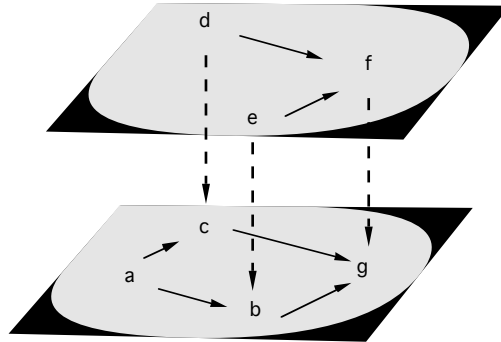


### 2.1.12 An example of two different arrows between the same nodes corresponding to two different proofs (in natural deduction) of $A \vdash B$ from $A \vdash B$

The graphs used in figures 2.1.4 and 2.1.5 are intuitively related through an operation that adds detail (or forgets, depending on the point of view). That is, graphs have a "social life" of their own that is useful to know about. Such relationships between graphs are called graph homomorphisms:

### 2.1.13 DEFINITION – graph homomorphism

A *homomorphism* of graphs  $\square: G \rightarrow H$  is a pair of maps  $\square_0: G_0 \rightarrow H_0$  and  $\square_1: G_1 \rightarrow H_1$  such that for each arrow  $f: x \rightarrow y$  of  $G$  we have  $\square_1(f): \square_0(x) \rightarrow \square_0(y)$  in  $H$ . That is, nodes are mapped to nodes and arrows to arrows but preserving sources and targets.



### 2.1.14 EXAMPLE

There is a "canonical" homomorphism between proof systems and consequence systems that consists of the identity on nodes and collapses any non-empty set of proof arrows between two nodes into just one consequence arrow. That is, this homomorphism "forgets" the details of the proofs, retaining just the fact that one exists to justify the consequence relation. For instance, when applied to the example in figure 2.1.12 it would deliver just one arrow from  $A \vdash B$  to  $A \vdash B$  as in 2.1.10.

Notice that the relationship between the inheritance graph used in figure 2.1.5 and the corresponding graph of feature sets and renamings (2.1.6) cannot be captured by a homomorphism because the source and target of arrows is reversed. Absence of a homomorphism in these circumstances seems to be accidental because the direction of the arrows is somewhat arbitrary: we could well have chosen a graph with the same collections of nodes and arrows but reversing the directions of the latter.

### 2.1.15 DEFINITION – duality

The graph obtained from another one by reversing the direction of arrows is called its *dual*: i.e. the dual of  $G = \langle G_0, G_1, src, trg \rangle$  is  $G^{op} = \langle G_0, G_1, trg, src \rangle$ . A homomorphism from a graph  $G$  to the dual  $H^{op}$  of a graph  $H$  is said to be *contravariant* between  $G$  and  $H$ .

An essential step towards the definition of what a category is concerns paths in graphs, i.e. what supports the move from direct to more "global" social relationships.

### 2.1.16 DEFINITION – path in a graph

Let  $G$  be a graph and  $x, y$  nodes of  $G$ . A *path* from  $x$  to  $y$  of length  $k > 0$  is a sequence  $f_1 \dots f_k$  of arrows of  $G$  (not necessarily distinct) such that:

- $src(f_1) = x$
- $trg(f_i) = src(f_{i+1})$  for  $1 \leq i \leq k-1$
- $trg(f_k) = y$

For every  $x$ , the path of length 0 at  $x$  (the empty path at  $x$ ) from  $x$  to  $x$  is, by convention, the empty sequence.

The collection of paths of  $G$  of length  $k$  is denoted by  $G_k$ . Hence:

- $G_0$  corresponds to the collection of nodes
- $G_1$  corresponds to the collection of arrows
- $G_2$  corresponds to the collection of pairs of composable arrows

## 2.2 Categories

Categories provide an abstraction over graphs by making paths the basic working elements – what are called *morphisms*; paths provide richer information about "social life" than just one-to-one relationships. For that purpose, categories add to graphs an identity map that "converts" nodes to morphisms, and a composition law on morphisms that internalises path construction. Morphism composition is required to be associative as for path concatenation: morphisms have no internal structure that can be derived from the order of composition.

### 2.2.1 DEFINITION – category

A *category*  $\mathbf{C}$  is a triple  $\langle G, \cdot, id \rangle$  where:

- $G$  is a graph

- $;$  is a map from  $G_2$  into  $G_1$  (called the composition law)
- $id$  is a map from  $G_0$  into  $G_1$  (called the identity map)

such that:

- $src(f;g)=src(f)$
- $trg(f;g)=trg(g)$
- $(f;g);h=f;(g;h)$
- $src(id_x)=trg(id_x)=x$
- for each  $f:x \square y$  of  $\mathbf{G}$ ,  $id_x;f=f$ ;  $id_y=f$

The nodes (resp. arrows) of  $G$  are also called the *objects* (resp. *morphisms*) of  $\mathbf{C}$ . The collection of objects of  $\mathbf{C}$  is denoted by  $|\mathbf{C}|$ . We will often use the notation  $c:\mathbf{C}$  to indicate that  $c$  is a object of  $\mathbf{C}$ , or a  $\mathbf{C}$ -object. Given objects  $x$  and  $y$ ,  $\text{Hom}_{\mathbf{C}}(x,y)$  denotes the collection of morphisms from  $x$  to  $y$ .

### 2.2.2 EXAMPLE – "the" category of sets

The category **SET**: objects are sets, morphisms are (total) functions between them, composition is functional composition – i.e.  $(f;g)(x)=g(f(x))$  – and the identity map assigns to every set the identity function on that set. Because function composition is associative and the identity map is both a left and right identity for function composition, all the conditions are met.

Function composition is usually denoted by the symbol  $\circ$  and the order of the arguments reversed, i.e. given  $f:x \square y$  and  $g:y \square z$ , the composed function  $f;g$  is also denoted by  $g \circ f$ . This is the the "application order":  $g \circ f(a)=g(f(a))$  for every  $a \square x$ . Most textbooks on Category Theory adopt this alternative notation for the composition law. Ultimately, the choice between one notation and the other is a matter of taste or convenience<sup>6</sup>. Our choice was motivated by the fact that it is closer to the diagrammatic notation (arrow sequencing) and, hence, supports the diagrammatic forms of reasoning that normally appeal to software engineers. Besides, as evidenced by the definition, the application order derives too much from set membership, i.e. reasoning with it is normally too close to the traditional set-theoretic approach from which we are trying to get away.

### 2.2.3 EXAMPLE – graphs

The category **GRAPH** has graphs as objects and its morphisms are the graph homomorphisms. The composition law is defined as follows: for every pair  $\square$  and  $\square$  of graph homomorphisms such that  $src(\square)=trg(\square)$ ,  $(\square;\square)_0=(\square_0;\square_0)$  and  $(\square;\square)_1=(\square_1;\square_1)$ . The identity map is defined as follows: for every graph  $G$ ,  $(id_G)_0=id_{G_0}$  and  $(id_G)_1=id_{G_1}$ .

<sup>6</sup> In fact, this was one of the major decisions that had to be made before starting writing! Many people are put off reading a book because they are used to the "other" notation. Hence, ultimately, the decision was made taking into account the intended audience (and for "pedagogical" reasons as explained).

PROOF

By taking graphs as nodes and graph homomorphisms as arrows, we do obtain a graph. The identity and associativity properties are inherited for each component (node and arrow) from the corresponding properties of functions between sets.

#### 2.2.4 EXAMPLE – logical implication

The category **LOGI** has as objects sentences, and morphisms between sentences correspond to the existence of a logical implication.

PROOF

In this category there is at most one morphism between two objects. So, the identity and composition equations are trivially satisfied. We just have to prove the existence of endomorphisms (reflexivity) and a composition law (transitivity). But  $(A \vdash A)$  is a tautology, and, from  $(A \vdash B)$  and  $(B \vdash C)$ , we can conclude  $(A \vdash C)$  (cut rule)!

In fact, every pre-order defines a category:

#### 2.2.5 PROPOSITION – pre-orders

Every pre-order  $\langle S, \leq \rangle$ , i.e., every set equipped with a reflexive and transitive relation, defines a category  $\mathbf{S}_\leq$  as follows:

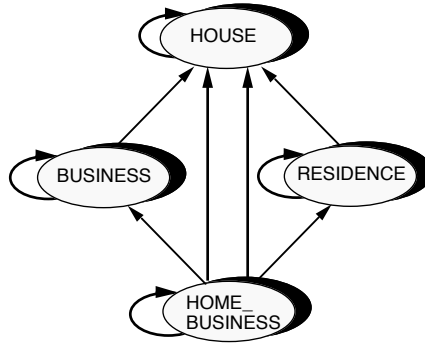
- $|\mathbf{S}_\leq| = S$  and there is a morphism between  $x$  and  $y$  in  $S$  iff  $x \leq y$
- composition is defined by applying the transitivity law of the pre-order
- identity morphisms are defined by applying the reflexivity laws of the pre-order

#### 2.2.6 EXAMPLE – proofs

The category **PROOF** has sentences as objects, and morphisms are proofs, i.e., a morphism  $f: A \sqsupset B$  is a specific proof of  $B$  from  $A$ . The identity morphisms are the trivial proofs of sentences from themselves (empty proofs). Proof composition corresponds to sequence concatenation – the cut rule.

#### 2.2.7 EXAMPLE – inheritance hierarchies

In Eiffel, the relationship "ancestor" is defined as the "reflexive and transitive closure" of the inheritance hierarchy: class  $A$  is an ancestor of class  $B$  iff  $A$  is  $B$  itself or  $A$  is an ancestor of a parent of  $B$  (i.e. of a class from which  $B$  inherits) [88]. Given an inheritance graph  $G$  between classes, e.g. example 2.1.3, the category **ancestor**( $G$ ) is generated from the graph by completing it with the arrows that result from reflexivity (identities) and transitivity (compositions).



### 2.2.8 PROPOSITION – category generated from a graph

Every graph  $G$  generates a category  $\mathbf{cat}(G)$ <sup>7</sup> whose objects are the nodes and whose morphisms are the paths of the graph. Identities are empty paths. Composition is concatenation of paths.

### 2.2.9 EXAMPLE – runs

Every state transition system generates the category of its runs.

We have already mentioned that morphisms define the structural aspects of the objects that the category makes available, what we have called their "social lives". We close this section with an example in which the notion of "preservation of structure" is, perhaps, more obvious. Nevertheless, we shall return to this topic in a later section in order to systematise some of the observations that we shall put forward in this example.

### 2.2.10 EXAMPLE – automata

A (deterministic) automaton consists of an input set  $X$ , a state set  $S$ , an output set  $Y$ , a transition function  $f: X \times S \rightarrow S$ , an initial state  $s_0 \in S$ , and an output function  $g: S \rightarrow Y$ . A notion of morphism of automata must be able to capture this structure by indicating how a given automaton can be simulated by another. Namely, it must translate the input, state and output sets from one automaton to the other in such a way that the transition functions, the initial states and the output functions of the two automata "agree".

More concretely, we should require of a morphism  $A \rightarrow A'$  that:

- if we perform a transition in  $A$  and map the resulting state into  $A'$ , we should get the state that is obtained by first translating the input and initial state into  $A'$  and then applying the transition function of  $A'$ ;
- the initial state of  $A$  should be mapped to the initial state of  $A'$ ;

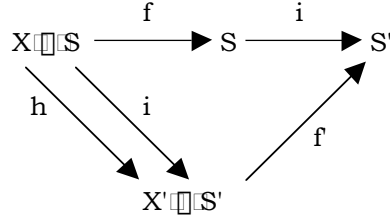
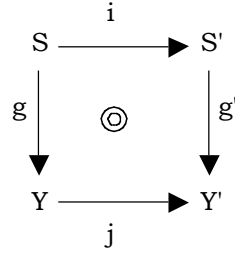
<sup>7</sup> The observant reader may have noted that we have already departed from our convention of using only upper case characters for the names of categories! This is not a random aberration, but reflects the fact that generating a category from a graph is a map from one category to another. Such maps, called functors, will be introduced in Chapter 5 and denoted using bold, lower case characters.



- the translation into  $A'$  of the output of a state of  $A$  should consist of the output in  $A'$  of the translation into  $A'$  of the original state of  $A$ .

Hence, a morphism from  $A = \langle X, S, Y, s_0, f, g \rangle$  to  $A' = \langle X', S', Y', s'_0, f', g' \rangle$  is defined to consist of three functions  $\langle h: X \rightarrow X', i: S \rightarrow S', j: Y \rightarrow Y' \rangle$  such that

- $f'i = \langle h, i \rangle; f'$
- $i(s_0) = s'_0$
- $g'j = i; g'$



The composition law is defined to be that of functions between sets, i.e. composition applies internally to each of the three components of the morphisms. Likewise, the identity for an automaton  $A$  is defined to be the triple that consists of the three identity functions over the sets of input, states and outputs.

The proof that a category – **AUTOM** – is indeed obtained in this way is very revealing of a procedure that will be systematised in section 3.2. Because the associativity of the composition law and the properties of the identity are automatically inherited from the corresponding properties of functions, all that needs to be proved is that (1) the composition of two morphisms is, indeed, a morphism (i.e. it satisfies the three equations above) and (2) the identity is, indeed, a morphism (satisfies the same three equations)! We prove just the case of a composition  $\langle h, i, j \rangle; \langle h', i', j' \rangle = \langle h; h', i; i', j; j' \rangle$  of morphisms  $\langle X, S, Y, s_0, f, g \rangle \rightarrow \langle X', S', Y', s'_0, f', g' \rangle$  and  $\langle X'', S'', Y'', s''_0, f'', g'' \rangle$ :

- $f; (i; i') = (f; i); i' = \langle h, i \rangle; f'; i' = \langle h, i \rangle; (f'; i') = \langle h, i \rangle; \langle h', i' \rangle; f'' = \langle h, i \rangle; \langle h', i' \rangle; f'' = \langle h; h', i; i' \rangle; f''$
- $(i; i')(s_0) = i'(i(s_0)) = i'(s'_0) = s''_0$
- $g; (j; j') = (g; j); j' = (i; g'); j' = i; (g'; j') = i; (i'; g'') = (i; i'); g''$

We have already mentioned that Category Theory supports and encourages forms of diagrammatic reason, this being one of the reasons why it is consistent with the modern culture in Software practice. Contrarily to what normally happens in Software Engineering, the notion of diagram is formal and the reasoning that can be done with diagrams is mathematical.

### 2.2.11 DEFINITION – diagram; commutative diagram

Let  $\mathbf{C}$  be a category and  $I$  a graph. A *diagram* in  $\mathbf{C}$  with *shape*  $I$  is a graph homomorphism  $\square: I \rightarrow \text{graph}(\mathbf{C})$  where  $\text{graph}(\mathbf{C})$  is the underlying graph of  $\mathbf{C}$ .

A diagram  $\square$  is said to *commute* iff, for every pair  $x, y$  of nodes and every pair of paths  $w = u_1 \dots u_m$ ,  $w' = v_1 \dots v_n$  from  $x$  to  $y$  in graph  $I$ ,  $\square_{u_1} \dots \square_{u_m} = \square_{v_1} \dots \square_{v_n}$  holds in  $\mathbf{C}$ .

The homomorphism defines a labelling of the graph  $I$ . That is, a diagram in a category can be seen as a graph whose nodes are labelled with objects and the arrows are labelled with morphisms of that category. The property that a diagram commutes

establishes a set of equalities between arrows. Hence, diagrams and commutativity provide us with the ability of doing equational reasoning in a “visual” form, an advantage that has not been exploited yet in Software Engineering. See, however, [48] for a more developed use of these possibilities for mathematical reasoning. To indicate that a diagram commutes, we will decorate it with the symbol  $\odot$ .

## 2.3 Distinguished kinds of morphisms

There are several classes of morphisms that have special properties worth knowing about because they allow us to recognise situations in which standard results or constructions apply.

### 2.3.1 DEFINITION – isomorphism

Let  $\mathbf{C}$  be a category and  $x, y$  objects of  $\mathbf{C}$ . A morphism  $f: x \rightarrow y$  of  $\mathbf{C}$  is said to be an *isomorphism* iff there is a morphism  $g: y \rightarrow x$  of  $\mathbf{C}$  such that:  $f;g = id_x$  and  $g;f = id_y$ . In these conditions,  $x$  and  $y$  are said to be isomorphic.

### 2.3.2 EXAMPLE

1. In **SET**, a morphism is an isomorphism iff it is bijective.
2. In **LOGI**, two formulae are isomorphic iff they are logically equivalent.

### 2.3.3 EXERCISE

What about isomorphic objects in **PROOF**?

The morphism  $g$  mentioned in the definition above is clearly unique: given  $h: y \rightarrow x$  in the same circumstances, we have

$$\begin{aligned}
 h &= h;id_x && \text{property of the identity} \\
 &= h;(f;g) && \text{because } f;g=id_x \\
 &= (h;f);g && \text{associativity} \\
 &= id_y;g && \text{because } h;f=id_y \\
 &= g && \text{property of the identity}
 \end{aligned}$$

This morphism is called the *inverse* of  $f$ .

Any two objects  $x$  and  $y$  related by an isomorphism  $f: x \rightarrow y$  have a very important property: the class of morphisms from  $x$  (resp. into  $x$ ) is in one-to-one correspondence with the class of morphisms from  $y$  (resp. into  $y$ ). This one-to-one correspondence is established by composing any morphism  $f_a: x \rightarrow a$  (resp.  $g_a: a \rightarrow x$ ) with the inverse of  $f$  (resp.  $f$  itself). Because morphisms characterise the interactions that objects can hold with other objects (their “social lives”), what this property says is that isomorphic objects interact in essentially the same way. That is to say, isomorphic objects cannot be

distinguished by interacting with them. In any context where an object is used, it can be replaced by an isomorphic one by using the isomorphism and its inverse to re-establish the interconnections: any incoming arrow

$$a \xrightarrow{g_a} x$$

is replaced by

$$a \xrightarrow{g_a} x \xrightarrow{f} y$$

and any outgoing arrow

$$a \xleftarrow{f_a} x$$

is replaced by

$$a \xleftarrow{f_a} x \xleftarrow{g} y$$

Hence, it is usual to treat any two isomorphic objects as being "essentially" the same.

Again, we should point out that this property holds in so far the "social life" that characterises the category is concerned. For a different category over the same objects, revealing other social aspects of the same objects, the isomorphism may not carry through. For instance, in object-oriented software development, two objects may be isomorphic in so far as having the same interfaces and exhibiting the same behaviour at their interfaces, but may have completely different implementations.

There are other classes of morphisms that are important to mention, namely those that generalise well-known properties in set-theory like injective and surjective functions. The way this generalisation is made reveals a lot of the way Category Theory operates and how it relates to and differs from Set Theory. For instance, the typical characterisation of an injective function  $f: x \rightarrow y$  in Set Theory is: *for every  $a, b \in x$ ,  $f(a)=f(b)$  implies  $a=b$* . This characterisation uses set-membership, of course. In Category Theory, we have to replace it by interactions with the objects involved, what we have been calling "social life". We have already mentioned in the introduction that elements of sets can be identified with morphisms from singleton sets, an observation that we shall formalise in section 4.1, but which can remain at an intuitive level for the purpose of this discussion. Hence, the set-theoretic definition amounts to saying that, for any (total) functions  $a, b: \{*\} \rightarrow x$ ,  $a; f = b; f$  implies  $a = b$ . Under this format, an injective function can be characterised as not interfering with independent observations that are made on the source. The categorical characterisation consists precisely in generalising this definition to arbitrary morphisms instead of just elements.

What is even more interesting is the fact that the generalisation carries through, in a dual way that we will formalise in the next chapter, to the characterisation of surjective functions. Surjectivity is all about interference with the social life of the target of the arrow in the sense that a non-surjective function leaves open degrees of freedom for interactions through the target to diverge even if they agree when mediated by the arrow.

**2.3.4 DEFINITION – mono and epimorphisms**

Consider an arbitrary category  $\mathbf{C}$  and morphism  $f: x \rightarrow y$  in  $\mathbf{C}$ .

1.  $f$  is said to be a *monomorphism*, or a *mono*, or *monic*, iff, for every pair of morphisms  $g, h: z \rightarrow x$ ,  $g \circ f = h \circ f$  implies  $g = h$ .

$$\begin{array}{ccccc} z & \xrightarrow{g} & x & \xrightarrow{f} & y \\ & \xrightarrow{h} & & & \end{array}$$

2.  $f$  is said to be an *epimorphism*, or an *epi*, or *epic*, iff, for every pair of morphisms  $g, h: y \rightarrow z$ ,  $f \circ g = f \circ h$  implies  $g = h$ .

Monos and epis satisfy many of the properties that we know from injective and surjective functions. They are left here as exercises.

**2.3.5 EXERCISE**

Consider an arbitrary category  $\mathbf{C}$ .

1. Prove that isomorphisms are both epic and monic.
2. Prove that the composition of monos (resp. epis) is also a mono (resp. epi).
3. Prove that if  $f \circ g$  is monic (resp. epic), then so is  $f$  (resp.  $g$ ).

A final word of caution though. The analogy with Set-Theory cannot be carried too far: the converse of the first property does not hold for arbitrary categories! That is, not all morphisms that are both epic and monic are isomorphisms. This is because, to be an isomorphism, an arrow needs to have a left and a right inverse which, contrarily to what happens in Set Theory, is not guaranteed simply by being monic and epic.

**2.3.6 DEFINITION – split mono and epimorphisms**

Consider an arbitrary category  $\mathbf{C}$  and morphism  $f: x \rightarrow y$  in  $\mathbf{C}$ .

1.  $f$  is said to be a *split monomorphism*, or a *split mono*, or *split monic*, iff it admits a right inverse, i.e. iff there is  $g: y \rightarrow x$  such that  $f \circ g = id_x$ .
2.  $f$  is said to be a *split epimorphism*, or a *split epi*, or *split epic*, iff it admits a left inverse, i.e. iff there is  $g: y \rightarrow x$  such that  $g \circ f = id_y$ .

**2.3.7 EXERCISE**

Consider an arbitrary category  $\mathbf{C}$ .

1. Prove that every split mono (resp. split epi) is monic (resp. epic).
2. Prove that every morphism that is a split mono and a split epi is an isomorphism. In fact, prove that only one of the morphisms needs to be "split".