

5 FUNCTORS

5.1 The social life of categories

It should come to no surprise that we are also interested in the social life of categories and, therefore, need to define a corresponding notion of morphism. Indeed, we have said more than once that, when defining a category, we are capturing only a particular view or structure of the underlying objects. There may well be more than one such notion of structure for the same collection of objects. For instance, in the case of object-oriented development, we have already mentioned the structure that relates to the way objects can be interconnected into systems, and the view that captures refinement. There may also be relationships that we wish to study between two different domains, for instance Eiffel specifications and processes. Hence, we need a way of reasoning about relationships between categories.

Because categories are structured objects, more precisely, graphs with the additional structure given by the identity arrows and the composition law, the definition of morphisms between categories (functors) will now look "standard":

5.1.1 DEFINITION – functor

Let \mathbf{C} and \mathbf{D} be categories. A *functor* $\square: \mathbf{C} \rightarrow \mathbf{D}$ is a graph homomorphism (2.1.13) from the graph of \mathbf{C} into the graph of \mathbf{D} such that:

- $\square_1(f;g) = \square_1(f); \square_1(g)$ for each path fg in C_2
- $\square_1(id_x) = id_{\square(x)}$ for each x in C_0 .

5.1.2 EXAMPLES

1. For any category \mathbf{C} , the identity functor $id_{\mathbf{C}}: \mathbf{C} \rightarrow \mathbf{C}$ consists of the identity map on objects and the identity map on morphisms.
2. For any given functor $\square: \mathbf{C} \rightarrow \mathbf{D}$, its dual is $\square^{pp}: \mathbf{D}^{pp} \rightarrow \mathbf{C}^{pp}$ is defined by the graph homomorphism that operates the same mappings on nodes and arrows.
3. We define a functor **nodes**: **GRAPH** \rightarrow **SET** as follows:
 - **nodes**(G_0, G_1, src, trg) = G_0
 - **nodes**($\square: G \rightarrow H$) = $\square_0: G_0 \rightarrow H_0$
4. We define a functor **PROOF** \rightarrow **LOGI** by mapping sentences to themselves and every proof between two sentences to a logical implication between them. This

is an example of a *forgetful functor*: the details of the proof are forgotten; only the fact that there exists a proof remains.

5. Another example of a forgetful functor is **sign: $PRES_{LTL} \rightarrow SET$** mapping presentations and their morphisms to the underlying signatures and signature morphisms. A similar forgetful functor is obtained for **$THEO_{LTL}$** .
6. We define a functor **$ANCESTOR \rightarrow SET^{op}$** by mapping classes to the set of their features and inheritance relationships to the functions that rename the features. Notice that the target category is the dual of **SET** . This is because, as we have illustrated in figure 2.1.6, the edges of the inheritance graph and the corresponding graph of features have opposite directions. Functors of the form $\Gamma: \mathbf{C} \rightarrow \mathbf{D}^{op}$ (or $\Gamma: \mathbf{C}^{op} \rightarrow \mathbf{D}$) are sometimes called contravariant between **\mathbf{C}** and **\mathbf{D}** . "Normal" functors are then called covariant.

5.1.3 EXAMPLE – relating programs and specifications

In Computing, functors are often used to express relationships between different levels of abstraction, namely by providing a way of abstracting properties from representations. In this context, the abstractions are sometimes called "specifications" and the representation "programs". For instance, although we have worked with specifications of Eiffel classes and not with the programs that implement the methods of such classes, it is clear that such specifications are at a lower level of abstraction than the temporal specifications that we defined in section . For instance, the temporal specifications can be used to describe general properties of the possible behaviours of a system whereas Eiffel specifications are concerned with more specific operational aspects of the execution of actions, like their effects on the attributes.

The notion of a program satisfying a specification is typically formalised through a satisfaction relation SAT. When specifications consist of sets of logical formulas such as those defined in section 3.5, the collection of specifications that are satisfied by a given program is ordered by inclusion and has a maximum: the union of all the sets of properties satisfied by the program. In this case, we can assign to every program P a "canonical" specification **$spec(P)$** – its strongest specification.

We have also seen that morphisms between models of system behaviour can be used to capture notions of simulation or refinement. When **$spec$** is a functor, this means that the refinement relationship defined on "programs" is captured by the corresponding notion of refinement at the level of specifications. When specifications are given as theory presentations, this means that program refinement is property-preserving, which is what one usually expects from a refinement relation. Hence, there is a natural way in which the satisfaction relation between programs and specifications can be expected to be functorial.

As an example, consider the mapping **$spec$** from **$CLASS_SPEC$** to **$PRES_{FOLTL}$** , where by **$PRES_{FOLTL}$** we are denoting the category of theory presentations for the first-order extension of the temporal logic presented in 3.5.4⁸:

- the image of every class signature is itself. That is, the features of the class are taken as symbols of the vocabulary of the logic.

⁸ For simplicity, and because the properties of the categories for the propositional and first-order versions of temporal logic in which we are interested are the same, we omit the extension. The reader interested in the logic itself can consult [66] as well as [39] for its use in a categorical framework.

- for every routine r , its specification (pre, pos) is mapped to the temporal formula

$$(r \sqcap pre \sqcap att \quad \mathbf{X}pos^*)$$

By att we denote the conjunction

$$\bigwedge_{a \sqcap att(\square)} (a = x_a)$$

where the variables x_a are all new, and by pos^* we denote the formula that is obtained from the expression pos by replacing every occurrence of every expression $(old\ a)$, where a is an attribute, by the variable x_a .

- the invariant I is mapped to itself.

For instance, the specification of the bank account given in 3.5.6 is mapped to the following set of temporal formulas:

$$\begin{aligned} deposit(i) &\sqcap balance = x_{balance} \quad \mathbf{X}(balance = x_{balance} + i) \\ withdrawal(i) &\sqcap balance \geq i \sqcap balance = x_{balance} \quad \mathbf{X}(balance = x_{balance} - i) \\ vip &\quad balance \geq 1000 \end{aligned}$$

To show that we obtain, indeed, a functor, we have to show in particular that morphisms of class specifications are property preserving. Consider a morphism $F: e = \langle \square, P, I \rangle \sqcap e' = \langle \square', P', I' \rangle$ of class specifications.

- Given any routine r of \square , its image $F(r)$ is such that its specification (pre', pos') is mapped to: $(F(r) \sqcap pre' \sqcap att' \quad \mathbf{X}pos^*)$. We know that

$$\begin{aligned} F(pre) &\vdash pre' && \text{because } F \text{ is a morphism} \\ pos' &\vdash F(pos) && \text{because } F \text{ is a morphism} \\ att' \quad \mathbf{X}pos^* &\vdash F(att) \quad \mathbf{X}F(pos^*) && \text{from } pos' \vdash F(pos) \text{ and the abbreviations} \end{aligned}$$

from which we can conclude that $(F(r) \sqcap pre' \sqcap att' \quad \mathbf{X}pos^*) \vdash F(r \sqcap pre \sqcap att \quad \mathbf{X}pos^*)$

- By definition of morphism, we also have $I' \vdash F(I)$

Hence, every axiom of **spec**(\square) is translated through F to a theorem of **spec**(\square').

It should come to no surprise that even categories can be organised in a category:

5.1.4 DEFINITION/PROPOSITION – the category of all categories

1. Let $\square: \mathbf{C} \rightarrow \mathbf{D}$ and $\square': \mathbf{D} \rightarrow \mathbf{E}$ be functors. By $\square'; \square$ we denote the functor defined by $(\square'; \square)_0 = \square'_0; \square_0$ and $(\square'; \square)_1 = \square'_1; \square_1$. This law of composition is associative and admits identities as defined in 5.1.2.1.
2. We can thus define the category **CAT** whose objects are the categories and whose morphisms are the functors.

PROOF

The proofs are trivial. However, bear in mind that, in defining **CAT**, there are problems of "size" in the same way as we alerted at the beginning of section 2.1. Again, see a more "mathematical-oriented" book on Category Theory, e.g. [79], for such matters.

Having a notion of morphism between categories, we can apply to categories and functors all the machinery that we have so far presented for manipulating objects and their morphisms. For instance, the notion of product that we defined in section 3.1 corresponds to a universal construction in the category **CAT**.

5.1.5 DEFINITION/PROPOSITION

Given categories **C** and **D**, we define functors $\pi_C: \mathbf{C} \times \mathbf{D} \rightarrow \mathbf{C}$ and $\pi_D: \mathbf{C} \times \mathbf{D} \rightarrow \mathbf{D}$, called the projections of the product, by mapping objects and morphisms of $\mathbf{C} \times \mathbf{D}$ to their components in **C** and **D**, respectively. These functors satisfy the universal property of functors in the following sense: given any category **E**, and functors $\pi_C: \mathbf{E} \rightarrow \mathbf{C}$ and $\pi_D: \mathbf{E} \rightarrow \mathbf{D}$, there is a unique functor $\pi: \mathbf{E} \rightarrow \mathbf{C} \times \mathbf{D}$ such that $\pi_C = \pi; \pi_C$ and $\pi_D = \pi; \pi_D$. We normally denote π by $\langle \pi_C, \pi_D \rangle$.

We can also define a product over functors:

5.1.6 DEFINITION/PROPOSITION

Given functors $\pi_1: \mathbf{C}_1 \rightarrow \mathbf{D}_1$ and $\pi_2: \mathbf{C}_2 \rightarrow \mathbf{D}_2$, their product $\pi_1 \times \pi_2: \mathbf{C}_1 \times \mathbf{C}_2 \rightarrow \mathbf{D}_1 \times \mathbf{D}_2$ is defined on objects by $(\pi_1 \times \pi_2) \langle c_1, c_2 \rangle = \langle \pi_1(c_1), \pi_2(c_2) \rangle$ and on morphisms by $(\pi_1 \times \pi_2) \langle f_1, f_2 \rangle = \langle \pi_1(f_1), \pi_2(f_2) \rangle$.

Functors come in all shapes and colours and, as morphisms between categories, they provide the means for characterising the structural properties of categories in the way they relate to other categories. Hence, it is useful to study the properties of functors and the way they allow us to reveal the structure of categories. We start with some elementary properties that result from the "functional" nature of functors as mappings between the sets of nodes and arrows.

5.1.7 DEFINITION

Let $\pi: \mathbf{C} \rightarrow \mathbf{D}$ be a functor.

1. π is called an *isomorphism* iff there is a functor $\pi': \mathbf{D} \rightarrow \mathbf{C}$ such that $\pi; \pi' = id_{\mathbf{C}}$ and $\pi'; \pi = id_{\mathbf{D}}$. In this case, **C** and **D** are said to be *isomorphic*.
2. π is called an *embedding* iff $\pi_!$ is injective, i.e. π is injective on morphisms.
3. π is said to be *faithful* iff all the hom-set restrictions $\pi_!: hom_{\mathbf{C}}(x, y) \rightarrow hom_{\mathbf{D}}(\pi_0(x), \pi_0(y))$ are injective.
4. π is said to be *full* iff all the hom-set restrictions $\pi_!: hom_{\mathbf{C}}(x, y) \rightarrow hom_{\mathbf{D}}(\pi_0(x), \pi_0(y))$ are surjective.

5.1.8 EXERCISE

Show that faithful functors are not necessarily embeddings: only when they are also injective on objects.

5.1.9 PROPOSITION

Let $\mathcal{I}:\mathbf{C}\rightarrow\mathbf{D}$ and $\mathcal{J}:\mathbf{D}\rightarrow\mathbf{E}$ be functors.

1. If \mathcal{I} and \mathcal{J} are isomorphisms (resp. embeddings, faithful, full), so is $\mathcal{J}\circ\mathcal{I}$.
2. If $\mathcal{J}\circ\mathcal{I}$ is an embedding (resp. faithful), so is \mathcal{I} .
3. If $\mathcal{J}\circ\mathcal{I}$ is full, so is \mathcal{I} .

5.1.10 EXAMPLE

Every subcategory \mathbf{D} of a category \mathbf{C} defines an inclusion functor $\mathcal{I}_{\mathbf{D},\mathbf{C}}:\mathbf{D}\rightarrow\mathbf{C}$. This functor is an embedding, and is full iff \mathbf{D} is a full subcategory of \mathbf{C} .

Embeddings are, intuitively, the best approximations to subcategories. In fact, in a world in which concepts are normally taken up to isomorphism, it seems intuitive not to make any difference between embeddings and inclusions of subcategories because all that is at stake are the "identities" of the objects involved, not their properties or structure. This intuition is supported by the following result:

5.1.11 PROPOSITION

A functor $\mathcal{I}:\mathbf{D}\rightarrow\mathbf{C}$ is a (full) embedding iff there is a (full) subcategory \mathbf{C}' of \mathbf{C} and an isomorphism $\mathcal{I}:\mathbf{D}\rightarrow\mathbf{C}'$ such that $\mathcal{I}=\mathcal{I}_{\mathbf{C}',\mathbf{C}}\circ\mathcal{I}$.

The relationships that we studied, in section 3.3, between subcategories and isomorphisms extend to functors:

5.1.12 PROPOSITION

Let $\mathcal{I}:\mathbf{C}\rightarrow\mathbf{D}$ be a functor.

1. \mathcal{I} *preserves isomorphisms*, i.e. if $f:x\rightarrow y$ is a \mathbf{C} -isomorphism, then $\mathcal{I}(f)$ is also an isomorphism.
2. if \mathcal{I} is faithful and full, then it *reflects isomorphisms*, i.e. if $\mathcal{I}(f)$ for $f:x\rightarrow y$ is an isomorphism, then f is itself an isomorphism.

The notions of preservation and reflection will be extended in 5.2.1 to universal constructions.

We end this section with the definition of a class of functors that arise from (co)reflective subcategories.

5.1.13 DEFINITION/PROPOSITION – reflector

Let \mathbf{D} be a reflective subcategory of a category \mathbf{C} . We define a functor $\mathcal{I}:\mathbf{C}\rightarrow\mathbf{D}$ as follows:

- every \mathbf{C} -object c has a \mathbf{D} -reflection arrow $\mathcal{I}_c:c\rightarrow d$. We define $\mathcal{I}(c)=d$;

5. FUNCTORS

- consider now a morphism $h:c \rightarrow c'$. The composition $h;\square_c$ is such that the definition of **D**-reflection arrow for c guarantees the existence and uniqueness of a morphism $h':\square(c) \rightarrow \square(c')$ such that $h;\square_c = \square_{c'}h'$. We define $\square(h)=h'$.

$$\begin{array}{ccc}
 c & \xrightarrow{\square_c} & \square(c) \\
 \downarrow h & \odot & \downarrow \square(h) \\
 c' & \xrightarrow{\square_{c'}} & \square(c')
 \end{array}$$

This functor is called a *reflector* for **C**, more precisely for the inclusion functor $\square_{\mathbf{C}}:\mathbf{D} \rightarrow \mathbf{C}$.

PROOF

We have to prove that a functor is indeed defined.

- The fact that a graph homomorphism is defined (i.e. sources and targets of arrows are respected) is trivially checked.
- By definition, $\square(id_c)$ is the unique **D**-morphism $h':\square(c) \rightarrow \square(c)$ satisfying $id_c;\square_c = \square_{c'}h'$. Because $id_{\square(c)}$ also satisfies that equation, we get $id_{\square(c)} = \square(id_c)$.
- Consider now the composition law.

$$\begin{array}{ccc}
 c & \xrightarrow{\square_c} & \square(c) \\
 \downarrow h & \odot & \downarrow \square(h) \\
 c' & \xrightarrow{\square_{c'}} & \square(c') \\
 \downarrow h' & \odot & \downarrow \square(h') \\
 c'' & \xrightarrow{\square_{c''}} & \square(c'')
 \end{array}$$

The composition $\square(h);\square(h')$ is such that $(h;h');\square_c = \square_{c''}(\square(h);\square(h'))$. This equality can be obtained from the equations satisfied individually by $\square(h)$ and $\square(h')$. Hence, by definition, $\square(h);\square(h')$ is the morphism $\square(h;h')$.

By duality, we obtain the notion of *co-reflector*. Notice that we had already made use of the idea of co-reflector in section 3.3 to explain the intuitions behind co-reflective subcategories. When applied to the categories defined in 3.5.4 for temporal specifications, we obtain several functors that perform the closure of sets of axioms, both as reflectors (from **PRES** and from **SPRES** to **THEO**) and co-reflectors (from **PRES** both to **SPRES** and **THEO**). Notice in particular that the functor from **PRES** to **THEO** is both a reflector and a co-reflector.

This proposition also shows that the relationship between automata and reachable automata is functorial. The map that eliminates non-reachable states as defined after 3.3.7 is a co-reflector.

We will generalise these kinds of functors in section 7.2. They are particular cases of classes of functors that generate free or canonical structures, or approximate other kinds of structures.

5.2 Universal constructions vs functors

In chapter 4, we motivated the study of universal constructions like (co)limits in terms of the ability to capture the collective behaviour of systems of interconnected components. In the previous section, we have just seen how functors provide us the means to map and relate different levels of abstraction in system development or different aspects of system description. One of the obvious questions that arises concerns the ability of functors to relate such universal constructions when applied in the categories related by the functors.

For an example of what we mean, consider what we consider to be one of the most profound “recent” contributions that have been made in the area of Programming Languages and Models: the separation between “Computation” and “Coordination” [50]. The best introduction we know to this topic can be found in [3]. In a nutshell, this whole area of research evolves around the ability to separate what in systems are the structures responsible for the computations that are performed and the mechanisms that are made available for coordinating the interactions that may be required between them. A decade of research has shown that languages that support this separation of concerns can improve our ability to handle complex systems. Its importance for Software Architectures has led to very close synergies between the two areas and there is a lot of cross-fertilisation going on.

One of the challenges that we personally felt in this area was to provide a mathematical characterisation of this separation that is independent of the particular languages that have been developed for this paradigm so that, on the one hand, we could start a systematic study of its properties and relationships to other paradigms and, on the other hand, extend it and support it with tools. This is what we started to do in [35] through the use of Category Theory.

The basic idea of our approach is to model this separation by a forgetful functor $\mathbf{int}:\mathbf{SYS}\rightarrow\mathbf{INT}$ where the category \mathbf{SYS} stands for “systems”, i.e. for whatever representations (models, behaviours, specifications, etc) we are using for addressing systems as a whole, and the category \mathbf{INT} is intended to capture the mechanisms that, in these representations, are responsible for the coordination aspects. We shall refer to the objects of \mathbf{INT} as “interfaces” following the idea that interconnections should be established only on the basis of what systems make available for interaction with other systems (e.g. communication channels), not at the level of the computations that they perform.

An example that will help us clarify more precisely what we have in mind can be given in terms of the linear temporal specifications that we have introduced in section 3.5. Therein, we motivated the fact that we were modelling the behaviour of concurrent processes at the level of the actions that are provided by their public interfaces in the sense that every signature identifies a set of actions in which a process can engage itself. The axioms of a specification provide an abstraction of the computations (traces) that are performed locally through the properties that they satisfy. The idea is, then, to take signatures as interfaces and characterise the abil-

ity of linear temporal logic specifications to separate computation and coordination in terms of properties of the functor $\mathbf{PRES}_{LTL} \dashv \mathbf{SET}$ that maps presentations and their morphisms to the underlying signatures and signature morphisms.

Which properties should we require of \mathbf{int} to capture the proposed separation? Basically, we have to capture the fact that any interconnection of systems is established via their interfaces.

For instance, an important property is that \mathbf{int} should be faithful. This means that morphisms of programs should not induce more relationships between programs than those that can be captured through their underlying interfaces. That is to say, by taking into consideration the computational part, we should not get additional observational power over the external behaviour of systems.

Another important property concerns the way colimits are computed. We have already mentioned that the colimit of a diagram expressing how a system is configured in terms of simpler components and interconnections between them, returns a model of the global behaviour of the system and the morphisms that relate the components to the global system. If only the interfaces matter for establishing the required interconnections, then the colimit of the diagram of systems should be obtainable from the colimit of the underlying diagram of interfaces. In other words, if we interconnect system components through a diagram, then any colimit of the underlying diagram of *interfaces* should be able to be lifted to a colimit of the original diagram of system components.

This property can be stated more precisely as follows: given any diagram $\mathbf{dia} : I \rightrightarrows \mathbf{SYS}$ and colimit $(\mathbf{int}(S_i) \dashv C)_{i \in I}$ of $(\mathbf{dia}; \mathbf{int})$ there exists a colimit $(S_i \dashv S)_{i \in I}$ of \mathbf{dia} such that $\mathbf{int}(S_i \dashv S) = (\mathbf{int}(S_i) \dashv C)$. When a functor satisfies a property like this one we say that it *lifts colimits*.

This means that when we interconnect system components, any colimit of the underlying diagram of interfaces establishes an interface for which a computational part exists that captures the joint behaviour of the interconnected components. Notice that this property does not tell us how to construct the lift, it just ensures that it exists. We shall return to this point later on. This property is really about (non)-interference between computation and coordination: on the one hand, the computations assigned to the components cannot interfere with the viability (in the sense of the existence of a colimit) of the underlying configuration of interfaces; on the other hand, the computations assigned to the components cannot interfere in the calculation of the interface of the resulting system.

A kind of “inverse property” is also quite intuitive: that every interconnection of system components be an interconnection of the underlying interfaces. In particular, that computations do not make viable a configuration of system components whose underlying configuration of interfaces is not. This property is verified when \mathbf{int} *preserves colimits*: given any diagram $\mathbf{dia} : I \rightrightarrows \mathbf{SYS}$ and colimit $(S_i \dashv S)_{i \in I}$ of \mathbf{dia} , $(\mathbf{int}(S_i) \dashv \mathbf{int}(S))_{i \in I}$ is a colimit of $(\mathbf{dia}; \mathbf{int})$. These two properties together imply that any colimit in \mathbf{SYS} can be computed by first translating the diagram to \mathbf{INT} , then computing the colimit in \mathbf{INT} , and finally lifting the result back to \mathbf{SYS} .

So: does the (forgetful) functor $\mathbf{sign} : \mathbf{PRES}_{LTL} \dashv \mathbf{SET}$ satisfy these properties? The answer is “yes”, but we will defer the proof and the recipe for constructing colimits to the next section. This is because this functor is an instance of a class that captures many useful structures in Computing and, hence, worth studying on its own, which includes the properties that we have just discussed. We shall return to the issue of

separating “Computation” and “Coordination” in several other places in the book, including examples.

Summarising, it is useful to classify functors vis-à-vis the way they relate universal constructions in the source and target categories:

5.2.1 DEFINITION

A functor $\mathbb{F}: \mathbf{C} \rightarrow \mathbf{D}$

1. *preserves*
 - a colimit $p: \coprod c$ of a diagram $\mathbb{F}: \mathbf{I} \rightarrow \mathbf{C}$ iff the co-cone $\{\mathbb{F}(p_a): \mathbb{F}(\coprod_a) \rightarrow \mathbb{F}(c)\}_{a \in \mathbf{I}_0}$, denoted by $\mathbb{F}(p): \mathbb{F}(\coprod) \rightarrow \mathbb{F}(c)$, is a colimit of \mathbb{F} .
 - colimits of shape \mathbf{I} iff it preserves the colimits of all diagrams $\mathbb{F}: \mathbf{I} \rightarrow \mathbf{C}$.
 - colimits iff it preserves the colimits of any diagram in \mathbf{C} .
2. *lifts* colimits iff for any diagram $\mathbb{F}: \mathbf{I} \rightarrow \mathbf{C}$ and colimit $p': \coprod d$ of \mathbb{F} , there is a co-cone $p: \coprod c$ that is a colimit of \mathbb{F} and $p' = \mathbb{F}(p)$. The lift is *unique* (or \mathbb{F} is said to lift colimits uniquely) when there is a unique co-cone $p: \coprod c$ satisfying the two properties.
3. *reflects* colimits iff for any diagram $\mathbb{F}: \mathbf{I} \rightarrow \mathbf{C}$ and co-cone $p: \coprod c$, if $\mathbb{F}(p)$ is a colimit of \mathbb{F} , then p is a colimit of \mathbb{F} .
4. *creates* colimits iff for any diagram $\mathbb{F}: \mathbf{I} \rightarrow \mathbf{C}$ and colimit $p': \coprod d$ of \mathbb{F} , there is a unique co-cone $p: \coprod c$ such that $p' = \mathbb{F}(p)$ and, moreover, p is a colimit of \mathbb{F} .

Definitions 2, 3 and 4 extend to specific classes of colimits (sums, co-equalisers, etc) as illustrated for preservation. The dual notions have the obvious names.

The following exercise will help the reader understand more quickly the difference between all these notions:

5.2.2 EXERCISE

Prove that

1. Every functor that creates colimits also reflects them.
2. A functor creates colimits iff it reflects and lifts colimits uniquely.

Note that the requirement on reflecting colimits is essential in case 2. For instance, it is easy to prove that the (forgetful) functor **THEO**_{LT} **SET** that maps theories and their morphisms to the underlying signatures and signature morphisms lifts colimits uniquely but it is easy to see that it does not create them: the colimit will choose the minimal theory among the whole class of theories that have the given signature and give rise to a co-cone; this class is only singular when some inconsistency arises from the interconnections and/or the base theories.

We can also relate these properties to 4.4.7:

5.2.3 EXERCISE

Prove that if \mathbf{C} is finitely co-complete then $\llbracket \cdot \rrbracket_{\mathbf{C}} : \mathbf{D} \rightarrow \mathbf{Set}$ preserves finite colimits iff it preserves initial objects and pushouts.

More results and related notions can be found, for instance, in [1,22].