

7 ADJUNCTIONS

7.1 The social life of functors

Yes, even functors are entitled to have their own social lives... And they can be quite rich too. In this book, we will remain at the level of what is basic and indispensable for covering this last topic of our introduction to Category Theory: adjunctions. For that purpose, we will just provide a short introduction to what are normally accepted to be "the" morphisms between functors: natural transformations.

The best way of understanding what natural transformations consist of, and can be used for, is to look at functors as views that one has from one category into another and formulate the properties that characterise the "preservation" of such views.

7.1.1 EXAMPLE – two views of Eiffel class specifications

We have already seen how one can look into Eiffel classes from the point of view of temporal logic specification: this is the view that is provided through the functor **spec** that we defined in 5.1.3. This functor accounts both for the pre/post conditions of methods and the class invariants. However, one is often interested in a more higher-level view of the behaviour of object classes that is concerned with the global properties that can be observed from their interfaces, typically their functions and routines. For this particular view, the actual specification of the functionality of the methods is of little relevance; they account for "how" these global properties are achieved rather than just "what" they are. Therefore, it makes sense that we define another functor **obsv: CLASS_SPEC** \rightarrow **PRES_{FOLTL}** that offers the more abstract point of view. Formally, these observable properties can be defined as sentences that involve only routines and functions. Hence, given an Eiffel class specification $e = \langle \square, P, I \rangle$ we define

$$\mathbf{obsv}(e) = \langle \square, \{ \square \mid \square \text{ LTL}(\text{fun}(\square) \mid \text{rou}(\square)) \mid \square \vdash \square \} \rangle$$

where \square is the set of axioms of **spec**(e). The reader is invited to check that this mapping defines a functor, i.e. that class inheritance induces an interpretation between the corresponding global properties.

Given two such views of class specifications, how can we relate them?

Clearly, such a relationship has to be established on the basis of morphisms that, for each class specification e , relate **obsv**(e) and **spec**(e). By definition, observable properties are derivable from the full class properties, so there is a morphism of presenta-

tions between each **obsv**(e) and **spec**(e). However, in order to respect the structure that morphisms (inheritance between class specification) induce on both views, the way observable properties relate through inheritance must be "the same" as the full specifications relate between them.

7.1.2 DEFINITION – natural transformation

Given two functors $\square: \mathbf{D} \rightarrow \mathbf{C}$ and $\square': \mathbf{D} \rightarrow \mathbf{C}$, a natural transformation η from \square to \square' , denoted by $\square \xrightarrow{\eta} \square'$ or $\square \xRightarrow{\eta} \square'$, is a function that assigns to each object d of \mathbf{D} a morphism $\eta_d: \square(d) \rightarrow \square'(d)$ of \mathbf{C} such that, for every morphism $f: d \rightarrow d'$ of \mathbf{D} , the following square commutes, in which case we say that η_d is *natural in d* or that the *naturality condition* holds:

$$\begin{array}{ccc}
 \square(d) & \xrightarrow{\eta_d} & \square'(d) \\
 \square(f) \downarrow & \circlearrowleft & \downarrow \square'(f) \\
 \square(d') & \xrightarrow{\eta_{d'}} & \square'(d') \\
 & \eta_{d'} & \\
 \mathbf{C} & \xleftarrow{\square, \square'} & \mathbf{D}
 \end{array}$$

$\begin{array}{ccc} & d & \\ & \downarrow f & \\ & d' & \end{array}$

7.1.3 EXERCISE

Workout the example in full by defining and proving the properties of the natural transformation.

The most obvious example of a natural transformation is, naturally, the identity:

7.1.4 DEFINITION – identity

Given a functor $\square: \mathbf{D} \rightarrow \mathbf{D}$ the identity natural transformation id_\square assigns to each object d of \mathbf{D} the identity morphism $id_{\square(d)}$.

In the rest of this section, we are going to analyse some of the mechanisms and properties that are available for using natural transformations when reasoning about how functors relate. The main results that we need concern the way natural transformations compose and the mechanisms we have to act on them.

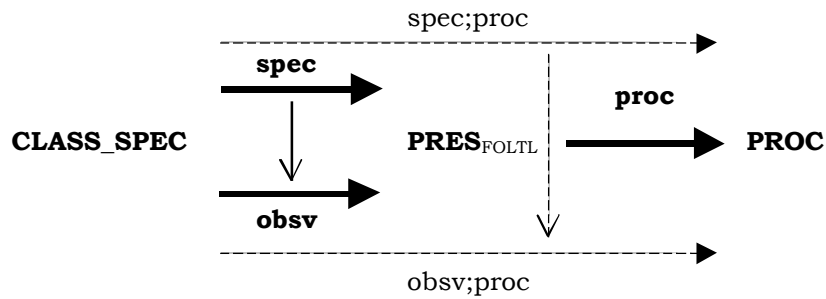
First, we can spell out what the dual of a natural transformation is:

7.1.5 DEFINITION

Consider two functors $\mathbb{C}:\mathcal{D} \rightarrow \mathcal{C}$ and $\mathbb{C}:\mathcal{D} \rightarrow \mathcal{C}$, and a natural transformation $\mathbb{C} \Rightarrow \mathbb{C}$. We define $\mathbb{C}^{op} \Rightarrow \mathbb{C}^{op}$ by $\mathbb{C}^{op}_d = \mathbb{C}_d$.

Notice that a morphism $\mathbb{C}^{op}(d) \rightarrow \mathbb{C}^{op}(d)$ in \mathcal{C}^{op} corresponds exactly to a morphism $\mathbb{C}(d) \rightarrow \mathbb{C}(d)$ in \mathcal{C} .

A useful class of operations on natural transformations is induced by functors into the sources, or from the targets, of the categories involved. For instance, supposing that we have a way (functor) to relate the domain of a viewpoint to another one (say between **PRES_{FOLTL}** and **PROC** as shown further down – 7.3.14), it makes sense to compose it with a natural transformation to provide a new one that extends the former to the second domain, e.g. to provide a mapping between the processes that capture the observable and the full view of object behaviour.



7.1.6 DEFINITION

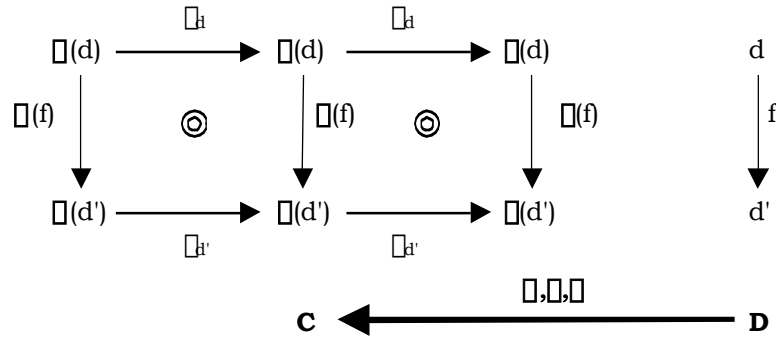
Consider two functors $\mathbb{C}:\mathcal{D} \rightarrow \mathcal{C}$ and $\mathbb{C}:\mathcal{D} \rightarrow \mathcal{C}$, and a natural transformation $\mathbb{C} \Rightarrow \mathbb{C}$.

1. Given $\mathbb{C}:\mathcal{E} \rightarrow \mathcal{D}$ we define $\mathbb{C} \Rightarrow \mathbb{C}$ by $(\mathbb{C})_e = \mathbb{C}_e$.
2. Given $\mathbb{C}:\mathcal{C} \rightarrow \mathcal{B}$ we define $\mathbb{C} \Rightarrow \mathbb{C}$ by $(\mathbb{C})_d = \mathbb{C}_d$.

These are "external" operations on a given natural transformation. An "internal" law can also be defined that allow us to compose simpler views into more complex ones in the sense that they bridge over sequences of viewpoints:

7.1.7 DEFINITION – composition

Consider three functors $\mathbb{C}, \mathbb{C}, \mathbb{C}:\mathcal{D} \rightarrow \mathcal{C}$ and natural transformations $\mathbb{C} \Rightarrow \mathbb{C}$ and $\mathbb{C} \Rightarrow \mathbb{C}$. The composition $\mathbb{C} \Rightarrow \mathbb{C}$ is defined by $(\mathbb{C})_d = \mathbb{C}_d \circ \mathbb{C}_d$.



7.1.8 EXERCISE

Prove that the composition of natural transformations is well defined, is associative and that the identities are, indeed, units for the composition law.

7.1.9 DEFINITION – natural isomorphism

A natural transformations $\varphi \circ \varphi'$ is said to be a *natural isomorphism* provided that each φ_i is an isomorphism, in which case φ and φ' are said to be *naturally isomorphic*, denoted by $\varphi \sim \varphi'$.

7.1.10 DEFINITION – equivalence of categories

Two categories \mathbf{C} and \mathbf{D} are *equivalent* when they admit functors $\varphi: \mathbf{D} \rightarrow \mathbf{C}$ and $\psi: \mathbf{C} \rightarrow \mathbf{D}$ such that $\varphi \circ \psi \sim id_{\mathbf{D}}$ and $\psi \circ \varphi \sim id_{\mathbf{C}}$.

Notice that equivalence between categories is a weaker notion than isomorphism (5.1.7). In particular, an equivalence does not operate up to "equality" but up to "isomorphism": for instance, given any object d of \mathbf{D} , $\varphi(\psi(d))$ does not need to be d but just isomorphic to d . Hence, to mark the fact, we do not use the term "inverse" for qualifying each of these functors with respect to the other but "pseudo-inverse". This is the terminology used, for instance, in [12].

For instance, for any institution, **PRES** and **THEO** are usually not isomorphic (the same theory may admit many presentations), but they are equivalent (all the presentations that define the same theory are isomorphic). This example shows that a category may be equivalent to one of its strict subcategories.

7.1.11 EXERCISE

What about **SPRES**?

Another example of an equivalence concerns two categories about which we have already highlighted many relationships in section 3 and 4:

7.1.12 EXAMPLE – equivalence between \mathbf{PAR} and \mathbf{SET}_\perp

The mappings

- $-_\perp$ that removes the designated element from pointed sets and transforms morphisms into partial functions
- $+_\perp$ that adds a new element to each set and completes partial functions by using the new element where they were undefined

define two functors whose compositions are naturally isomorphic to the identity.

Indeed, both categories are, basically, the "same" in the sense that one just makes explicit the partiality by presenting the designated element. In many applications to Computing, namely in the modelling of system behaviour as we have been illustrating with processes and specifications (theories), one tends to switch between one category and the other depending on whether we wish to attribute a meaning to the designated element, like for processes where it models steps performed by the environment, or be less "bureaucratic" (more pragmatic) and keep it just implicit. In fact, this is what we did in the graphical representations of the examples of universal constructions on processes: to simplify the notation, we omitted the designated element from the alphabets and represented the morphisms as partial functions. The "old" duality between "syntax" and "semantics" also tends to play a role: for semantic domains, like processes, it is useful to handle the designated element; but for syntax, like that of the parallel design language CommUnity that we study in chapter 8, it is often more "practical" to work instead with partial functions. The equivalence then tells us how we can switch between the two views.

Notice that the two categories are not isomorphic: the new element that is used to complete a set is not necessarily the one that was forgotten when that set is obtained from a pointed one. In fact, it is interesting for the reader to come up with a "real" definition of the functor that adds new elements to sets to make pointed sets: which elements does it add? Is there a "canonical" way of performing this completion?

The fact that we do not have an isomorphism is more significant for the "social life" of the categories themselves than for the structures that they endow internally on their objects (which is basically the same). We give an example of what we mean by this in 7.3.13.

7.2 Reflective functors

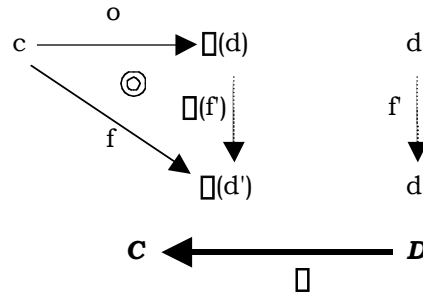
Keeping the promise of writing for the community of software scientists and practitioners, who are not necessarily as mathematically mature as most of the other books on Category Theory assume, we shall abstain from the traditional introduction to adjunctions, such as the construction of free monoids and other mathematical structures, or Galois connections. Instead, and besides giving examples closer to system development, we will follow the method adopted in previous chapters, and use concepts introduced about subcategories to motivate the definition of "similar" properties of functors, based on the fact that the inclusion of a subcategory in another one defines a functor.

The generalisation from subcategories into functors that interests us for adjunctions concerns reflections and co-reflections. You may wish to revisit 3.3.8–3.3.12 where we motivated (co)reflections as defining a specialised class of "secretaries" through which all the interactions can be factorised. The generalisation consists, once again, in replacing the inclusion by a functor; in a way, our secretaries become "interpreters", i.e. some kind of "adjuncts" through which all the communication with the "other side of the functor" is handled.

7.2.1 DEFINITION – reflection

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

1. Let c be a \mathbf{C} -object. A \square -reflection for c is a \mathbf{C} -morphism $o: c \rightarrow \square(d)$ for some \mathbf{D} -object d such that, for any \mathbf{C} -morphism $f: c \rightarrow \square(d')$ where d' is a \mathbf{D} -object, there is a unique \mathbf{D} -morphism $f': d \rightarrow d'$ such that $f = o \circ \square(f')$ i.e. the \mathbf{C} -diagram commutes:



2. The functor \square is said to be **reflective** iff every \mathbf{C} -object admits a \square -reflection. We tend to denote functors that are reflective with the special arrow $\bullet \rightarrow$.

That is to say, given an object c of \mathbf{C} , we are looking for the "best" object of \mathbf{D} that can handle its relationships "across the border", i.e. with other objects of \mathbf{D} through the functor \square . The morphism o can be seen as the protocol that c needs to have with its "interpreter", or the "distance" that remains to be bridged in \mathbf{D} .

Notice that \square -reflections are \square -structured morphisms in the sense of 6.2.2. The following proposition provides a useful characterisation of reflections:

7.2.2 PROPOSITION

1. Given a functor $\square: \mathbf{D} \rightarrow \mathbf{C}$ and a \mathbf{C} -object c , the \square -reflections for c are the initial objects of the category $c \downarrow \square$.
2. A functor $\square: \mathbf{D} \rightarrow \mathbf{C}$ is reflective iff, for every \mathbf{C} -object c , the category $c \downarrow \square$ has initial objects.

7.2.3 EXERCISE

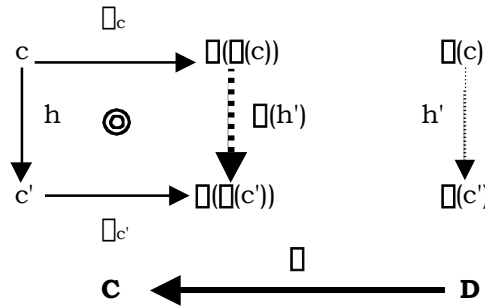
Prove 7.2.2 and conclude that \square -reflections for an object c are essentially unique, i.e. two \square -reflections for c are isomorphic and, if $f: c \rightarrow \square(d)$ is a \square -reflection for c and $h: d \rightarrow d'$ is an isomorphism, then $f \circ \square(h)$ is also a \square -reflection for c .

The notion of reflector for the inclusion functor defined by a reflective subcategory put forward in 5.1.13 can also be generalised to an arbitrary reflective functor.

7.2.4 DEFINITION/PROPOSITION – reflector

Let $\square: \mathbf{D} \rightarrow \mathbf{C}$ be a reflective functor. We define a functor $\square: \mathbf{C} \rightarrow \mathbf{D}$ as follows:

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



This functor is called a *reflector* for \square .

PROOF

The proof is trivially generalised from the one given in 5.1.13 and is left as an exercise.

7.2.5 DEFINITION/PROPOSITION – reflection unit

Let $\square: \mathbf{D} \rightarrow \mathbf{C}$ be a reflective functor and $\square: \mathbf{C} \rightarrow \mathbf{D}$ a reflector. The definition of \square provides directly a natural transformation $\text{id}_{\mathbf{C}} \xrightarrow{\square} \square; \square$. We call it the *unit* of the reflection.

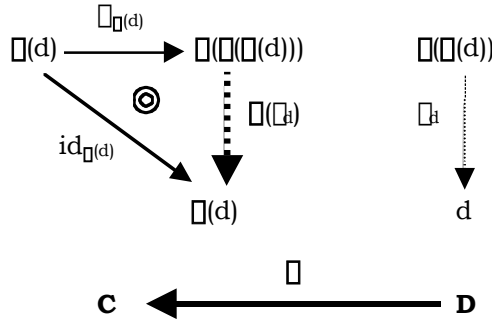
By duality, we obtain the notion of *co-reflective* functor and *co-reflector* for a co-reflective functor, generalising what was defined for co-reflective subcategories. We tend to denote functors that are co-reflective with the special arrow \longleftrightarrow .

7.2.6 PROPOSITION

Let $\square: \mathbf{D} \rightarrow \mathbf{C}$ be a reflective functor. Every reflector $\square: \mathbf{C} \rightarrow \mathbf{D}$ for \square is co-reflective and admits \square as a co-reflector. Moreover, given any \mathbf{D} -object d , its \square -co-reflection $\square_k: \square(\square(d)) \rightarrow d$ satisfies $\square_{\square(d)}; \square(\square_k) = \text{id}_{\square(d)}$.

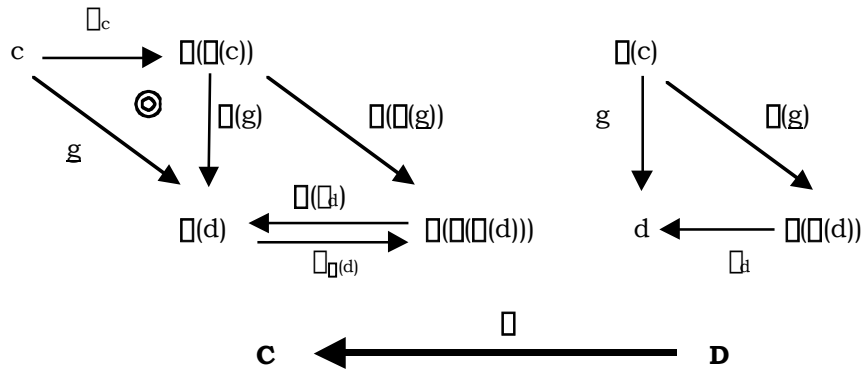
PROOF

Let d be a \mathbf{D} -object. The universal properties of $\square_{\square(d)}$ ensure the existence and uniqueness of a morphism $\square_k: \square(\square(d)) \rightarrow d$ such that $\square_{\square(d)}; \square(\square_k) = \text{id}_{\square(d)}$.



It is easy to see that η_d is, indeed, a η -co-reflection for d . Let $g: c \rightarrow d$ be a D -morphism. We are going to prove that $g = \eta_c; \eta(g); \eta_d$ satisfies $\eta(g); \eta_d = g$. Because there is only one morphism $h: c \rightarrow d$ such that $\eta_c; \eta(h) = g$, we are going to prove that $\eta(g); \eta_d$ satisfies that equation:

$$\begin{aligned} \eta_c; \eta(\eta(g); \eta_d) &= \eta_c; \eta(\eta(g)); \eta(\eta_d) \\ &= g; \eta_{\eta(d)}; \eta(\eta_d) \text{ because of the properties of natural transformations} \\ &= g \text{ because of the properties of } \eta_d \end{aligned}$$



Moreover, g is the only morphism $g': c \rightarrow d$ that satisfies $g = \eta(g'); \eta_d$ because the equation implies $\eta_c; \eta(g) = \eta_c; \eta(\eta(g')); \eta(\eta_d) = g'; \eta_{\eta(d)}; \eta(\eta_d) = g'$.

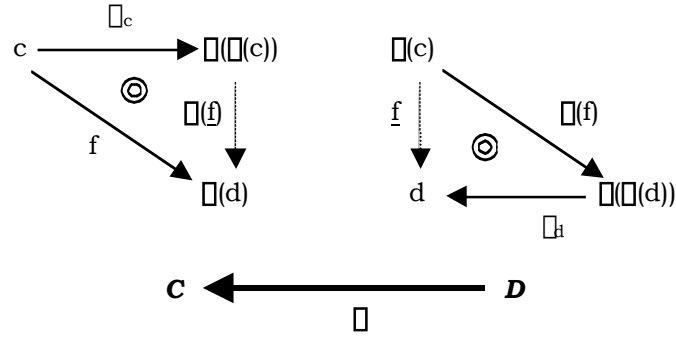
7.2.7 COROLLARY

Consider a reflective functor $\eta: D \rightarrow C$ and its reflector $\eta: C \rightarrow D$.

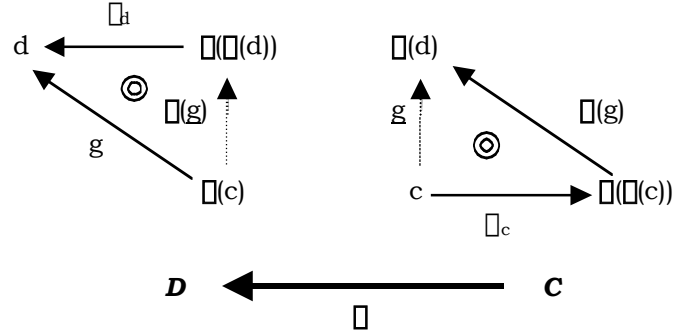
- From proposition 7.2.6 we derive a natural transformation $\eta; \eta \dashv \eta$ id_D that we call the *co-unit* of the reflection. The two natural transformations (unit and co-unit) satisfy

$$\begin{aligned} \eta \dashv \eta \dashv \eta &= \eta \dashv \eta \dashv \eta \\ \eta \dashv \eta \dashv \eta &= \eta \dashv \eta \dashv \eta \end{aligned}$$

- Every morphism $f: c \rightarrow d$ can be mapped to $f = \eta(f); \eta_d; \eta_c$ which is the unique morphism $\eta(c) \rightarrow d$ that makes the C -triangle commute.



and every morphism $g: \hat{C}(c) \rightarrow d$ can be mapped to $\hat{g} = \hat{C}_c; \hat{C}(g): \hat{C}(c) \rightarrow \hat{C}(d)$ which is the unique morphism $\hat{c} \rightarrow \hat{C}(d)$ that makes the \mathbf{D} -triangle commute.



These mappings define a bijection that is "natural" in \mathbf{C} and \mathbf{D} in the sense that it satisfies, for every $h: c' \rightarrow c$ and $k: d \rightarrow d'$,

$$\hat{C}(h); \hat{f}; k = \hat{h}; \hat{f}; \hat{C}(k) \text{ and } h; \hat{g}; \hat{C}(k) = \hat{C}(h); \hat{g}; k$$

PROOF

We leave it as an exercise. Notice that, for instance,

$$\hat{f} = \hat{C}_c; \hat{C}(f) = \hat{C}_c; \hat{C}(C(f); \hat{C}_d) = \hat{C}_c; \hat{C}(C(f)); \hat{C}(\hat{C}_d) = f; \hat{C}_{D(d)}; \hat{C}(D(d)) = f$$

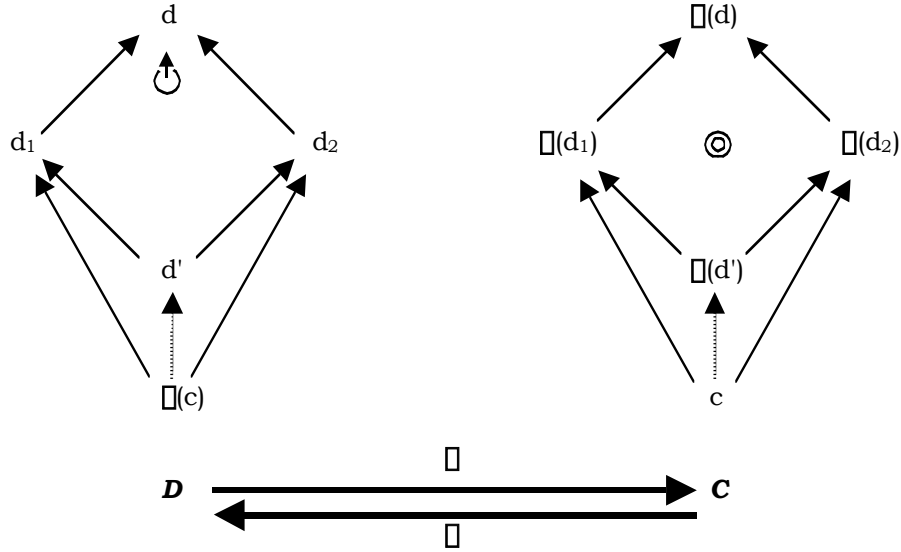
Some useful properties of reflective functors are:

7.2.8 PROPOSITION

1. Reflective functors compose, i.e. if $\hat{C}: \mathbf{E} \rightarrow \mathbf{D}$ and $\hat{D}: \mathbf{D} \rightarrow \mathbf{C}$ are reflective then so is $\hat{D}; \hat{C}: \mathbf{E} \rightarrow \mathbf{C}$.
2. Reflective functors preserve limits.

PROOF

Left as an exercise. We just give a brief illustration of 2 for the case of pullbacks. We start with a pullback diagram in \mathbf{D} that we translate to \mathbf{C} . If we now consider a commutative cone $\langle c \rightarrow \hat{C}(d_i) \rangle$, we can lift it back to \mathbf{D} through the reflection as a commutative cone $\langle \hat{C}(c) \rightarrow d_i \rangle$. From the properties of the pullback, we are given a morphism $\hat{C}(c) \rightarrow d$ of commutative cones that translates, once again, to \mathbf{C} as a morphism of commutative cones. Uniqueness can be easily checked.



The characterisation of reflective functors provided through 7.2.2 allows us to give examples among some of the constructions analysed in chapter 6:

7.2.9 COROLLARY

1. Let $\square: \mathbf{C} \rightarrow \mathbf{SET}$ be a functor. The functor $\square: \mathbf{spa}(\square) \rightarrow \mathbf{C}$ that forgets the **SET**-component of each object (6.3.3) is both reflective and co-reflective. The \square -reflection of any **C**-object c is $id_c: c \rightarrow \square(\langle c, \emptyset \rangle)$ and its co-reflection is $id_c: \square(\langle c, \square(c) \rangle) \rightarrow c$.
2. Consider an indexed category $\square^{I^p} \square \mathbf{CAT}$. The functor $\mathbf{fib}(\square): \mathbf{FLAT}(\square) \rightarrow \mathbf{I}$ that projects objects and morphisms to their **I**-components (6.4.7) is reflective iff, for every index i , $\square(i)$ has an initial object $0_{\square(i)}$, and is co-reflective iff, for every index i , $\square(i)$ has a terminal object $1_{\square(i)}$. The $\mathbf{fib}(\square)$ -reflection of any index i , when it exists, is $id_i: i \rightarrow \mathbf{fib}(\square)(\langle i, 0_{\square(i)} \rangle)$, and its $\mathbf{fib}(\square)$ -co-reflection, when it exists, is $id_i: \mathbf{fib}(\square)(\langle i, 1_{\square(i)} \rangle) \rightarrow i$.

Another obvious example of (co)reflective functors concerns, of course, (co)reflective subcategories:

7.2.10 COROLLARY

1. If **D** is a reflective subcategory of a category **C**, then the inclusion functor is reflective.
2. If **D** is a co-reflective subcategory of a category **C**, then the inclusion functor is co-reflective.

We can also generalise the results (3.3.10) that relate full (co)reflective subcategories with properties of the (co)unit:

7.2.11 PROPOSITION

Consider a reflective functor $\square: \mathbf{D} \rightarrow \mathbf{C}$ and let \square_i be its co-unit.

1. \square is faithful iff, for every \mathbf{D} -object d , \square_i is epi.
2. \square is full and faithful iff, for every \mathbf{D} -object d , \square_i is an isomorphism.

7.2.12 EXERCISE

Complete the constructions and proofs of 7.2.6 and 7.2.7 to get acquainted with these newly acquired tools¹⁰.

7.3 Adjunctions

The reader already acquainted with Category Theory will have noticed that we are not only following a different path to the topic, as already justified even if it turns out not to be that different from [1], but also departing from the standard terminology (if one really exists) for adjunctions. The reason is that the terminology that we are introducing is a natural continuation of the one we used for subcategories (and this one is standard, or at least it complies with [79], which comes more or less to the same). What we have called a \square -reflection for d is called in [1] a \square -universal arrow for d (or with domain d), and a reflective functor is called therein an *adjoint*. The prefix *co* is used in [1] exactly in the same way so that a \square -co-reflection is a \square -co-universal arrow and a co-reflective functor is co-adjoint.

Although we prefer the terminology that we introduced in the previous sections, there are also good reasons for using the terminology introduced in [1]: adjoints and co-adjoints arise in *adjunctions*. In this section, we are going to introduce the standard terminology on adjunctions (i.e. [79]) because it is *really* standard. Adjunctions are an intrinsic part of the vocabulary of Category Theory. What is hardly standard is the way to approach and define the notion of adjunction. This is where, as authors, we can allow ourselves a little illusion of originality.

7.3.1 DEFINITION – adjunction

An *adjunction* from a category \mathbf{C} to another category \mathbf{D} consists of

- two functors $\square: \mathbf{D} \rightarrow \mathbf{C}$ and $\square^*: \mathbf{C} \rightarrow \mathbf{D}$
- two natural transformations $\text{id}_{\mathbf{C}} \square^* \square \rightarrow \square^* \square$ and $\square \square^* \rightarrow \text{id}_{\mathbf{D}}$ satisfying

$$\square \square^* \square \rightarrow \square \square^* \square \rightarrow \square \square^* \square \rightarrow \square \square^* \square$$

$$\square \square^* \square \rightarrow \square \square^* \square \rightarrow \square \square^* \square \rightarrow \square \square^* \square$$

¹⁰ And do have fun exploring the symmetries revealed through the diagrams, sketching the odd duality... The more mathematically mature readers may wish to identify Galois connections in the process.

We use the notation $\mathcal{C} \xrightleftharpoons[\mathcal{D}]{\mathcal{A}}$ for such an adjunction.

Given an adjunction $\mathcal{C} \xrightleftharpoons[\mathcal{D}]{\mathcal{A}}$

- \mathcal{A} can be called: the right adjoint, the adjoint, the forgetful functor
- \mathcal{D} can be called: the left adjoint, the co-adjoint, the free functor
- η is called the unit
- ϵ is called the co-unit.

In what concerns the terminology, the left/right classification is quite widespread; (co)adjoints are used in [1] as already mentioned. Classifying the functors as forgetful/free can be every helpful when the roles that they play is obvious. This is precisely the case of the adjunctions that result from reflective subcategories, functor-structured categories and indexed categories as illustrated below: the (right) adjoint usually "forgets" part of the structure of objects that the left/co-adjoint is able to freely generate the additional structure.

An immediate example is:

7.3.2 PROPOSITION

Every equivalence defines two adjunctions.

It is useful to state explicitly a result about duality:

7.3.3 PROPOSITION

For every adjunction $\mathcal{C} \xrightleftharpoons[\mathcal{D}]{\mathcal{A}}$, $\mathcal{D}^{\text{op}} \xrightleftharpoons[\mathcal{C}^{\text{op}}]{\mathcal{A}^{\text{'}}}$ is also an adjunction (its dual).

And also about composition:

7.3.4 PROPOSITION

Given functors $\mathcal{A}:\mathcal{D} \rightarrow \mathcal{C}$, $\mathcal{B}:\mathcal{C} \rightarrow \mathcal{D}$, $\mathcal{C}:\mathcal{E} \rightarrow \mathcal{D}$, $\mathcal{D}:\mathcal{D} \rightarrow \mathcal{E}$, if \mathcal{A} is a left adjoint of \mathcal{B} and \mathcal{C} is a left adjoint of \mathcal{D} then $\mathcal{B}\mathcal{C}$ is a left adjoint of $\mathcal{D}\mathcal{A}$.

There are many alternative ways of characterising adjunctions. It can even be hard to find two books that adopt the same characterisation as the defining one. However, they all involve, in some way or the other, but not arbitrarily, the properties analysed in 7.2.7. For instance,

7.3.5 PROPOSITION

An adjunction from a category \mathcal{C} to another category \mathcal{D} can be obtained from

- two functors $\mathcal{A}:\mathcal{D} \rightarrow \mathcal{C}$ and $\mathcal{B}:\mathcal{C} \rightarrow \mathcal{D}$
- a bijection between morphisms $c \rightarrow \mathcal{A}(d)$ and $\mathcal{B}(c) \rightarrow d$ that is natural in \mathcal{C} and \mathcal{D} .

PROOF

Much of the proof has been sketched in 7.2.7. We leave it as an exercise to complete it.

This characterisation is useful for the following example in particular:

7.3.6 PROPOSITION

The power-set functor $2^{_} : \mathbf{SET}^p \rightarrow \mathbf{SET}_{_}$ that maps every set to its powerset as a pointed set, the empty set being the designated element, and every function to its inverse image, defines an adjunction from $\mathbf{SET}_{_}$ to \mathbf{SET}^p . Its left adjoint computes power sets of proper elements (i.e. excluding the designated element) and inverse images.

PROOF

The reader is invited to carry out the proof as it is very instructive if not challenging due to the fact that it operates on a contravariant functor! The natural bijection is defined by associating functions of the form $f: A \rightarrow 2^{_}$ with $g: B \rightarrow 2^{_}$ by $a \sqsubseteq g(b)$ iff $b \sqsubseteq f(a)$.

An adjunction is a very strong relationship between two categories: it allows us to give canonical approximations of objects in one domain with respect to the structural properties that are captured in the other domain. Examples of the use of adjunctions abound, even in the particular aspects of Computing Science that interest us in this book. One that we have worked out and presented in [36] concerns the synthesis of programs from specifications:

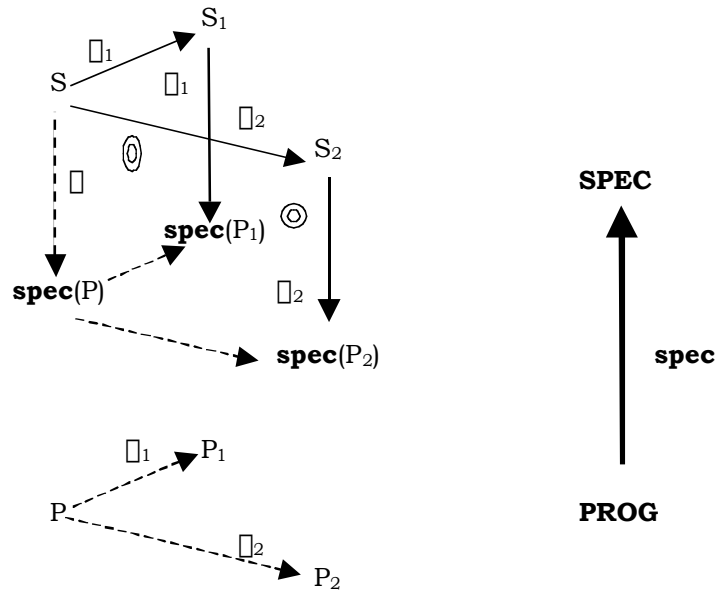
7.3.7 EXAMPLE – synthesis of programs from specifications

We have already mentioned (5.1.3) how the satisfaction relationship between programs and specifications may be captured, in certain circumstances, by a functor $\mathbf{spec}: \mathbf{PROG} \rightarrow \mathbf{SPEC}$ that maps every program to the maximal set of properties that it satisfies. In such situations, we have called every morphism $S \rightarrow \mathbf{spec}(P)$ a possible realisation of the specification S by the program P (6.2.1). In this context, we have illustrated how systems can be evolved by interconnecting components that make new required properties to emerge (6.1.24). We have also showed how the process of assembling a system from smaller components, including the interconnection of new components, can be addressed in a compositional way by addressing realisations and not just individual specifications or programs (6.2.4).

We are now interested in incorporating into the picture the ability to synthesise programs from specifications as a means of supporting the process of compositional evolution that we have been addressing and according to which the addition of a component to a system should not require the recalculation (or the re-synthesis) of the whole system but only of the new component and its interconnections to the previous system. To illustrate our purpose, consider once again the vending machine (3.5.6). In (6.1.24), we developed the specification of a regulator and showed that, once interconnected with the vending machine, the new system did not allow arbitrary sales of cigars but required the insertion of a special token before a cigar can be selected. Assuming that the original vending machine was implemented and run-

ning, and that we were in possession of a realisation of the regulator, possibly by having used the synthesis method of [85], we would like to be able to synthesise the interconnections between the two programs (the running vending machine and the realisation of the regulator) in order to obtain a realisation of the specification diagram.

In summary, given realisations of component specifications (either obtained through traditional transformational methods, or synthesised directly from the specifications, or reused from previous developments), we would like to be able to synthesise the interconnections between the programs in such a way that the program diagram realises the specification diagram. That is to say, given specifications S_1 and S_2 (newly) interconnected via morphisms $\square_1: S \square S_1$ and $\square_2: S \square S_2$, and realisations $\langle \square_1, P_1 \rangle$, $\langle \square_2, P_2 \rangle$ of S_1 and S_2 , respectively, one would like to be able to synthesise a realisation $\langle \square, P \rangle$ of S and morphisms $\square_1: P \square P_1$ and $\square_2: P \square P_2$ such that $\square; \text{spec}(\square_i) = \square_i; \square_i$ ($i=1,2$).



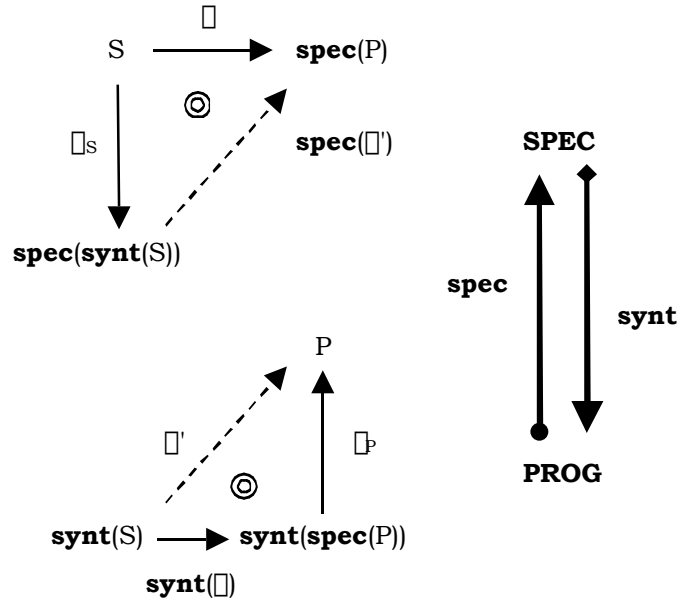
This general statement of what it means to synthesise interconnections makes it clear that it is both necessary to synthesise the middle program P and the morphisms \square_i that are required to interconnect the given programs. Because, in the general case, any object can be used in an interconnection, this suggests, rather obviously, that a functor **synt**: **SPEC** \square **PROG** is required that is somehow related to **spec**. One possible such relationship is for **spec** to be the inverse (5.1.7) of **synt**, but this is a rather strong property because it would require the two categories of programs and specifications to be isomorphic. Clearly, if this were to be the case, we could hardly claim that we were dealing with two different levels of abstraction. Hence, it makes sense to look for weaker properties of the relationship between the two functors.

It seems clear that, more than programs, synthesis must return realisations of the given specifications. That is to say, for every specification S , **synt**(S) must be provided together with a morphism $\square_S: S \square \text{spec}(\text{synt}(S))$ that establishes **synt**(S) as one of the possible realisations of S . Hence, \square_S expresses a correctness criterion for **synt**. Moreover, **synt** must respect interconnections in the following sense: given a specification diagram $\square: \mathbf{I} \square \mathbf{SPEC}$, it is necessary that the program diagram $\square; \text{synt}$ be a realisation of \square through $(\square_S)_{i \in \mathbf{I}}$ as defined in 6.2.4, i.e. we must have, for every arrow $f: i \square j$ in \mathbf{I} , $\square_S f \square \square_S = \square_S; \text{spec}(f)$. But these are the ingredients that define a natural transformation. Hence, **synt** must be provided together with a natural transformation $\square: \mathbf{1}_{\mathbf{SPEC}} \square \text{synt}; \text{spec}$.

Consider now the synthesis of interconnections themselves. Given an interconnection of specifications $\square:S \rightarrow \text{spec}(P)$ we should be able to synthesise $\square':\text{synt}(S) \rightarrow P$ in such a way that the interconnection is respected, i.e. $\square = \square_s \circ \text{spec}(\square')$. This is equivalent to defining a (natural) bijection between the morphisms $S \rightarrow \text{spec}(P)$ and the morphisms $\text{synt}(S) \rightarrow P$. But this is, precisely, the property that characterises the existence of an adjunction between **SPEC** and **PROG**. Hence, synthesis of interconnections can be characterised by the existence of a reflector (left adjoint) **synt** for **spec**. Notice that 7.2.2 characterises the synthesis functor precisely in terms of the existence, for every specification, of a "minimal" realisation in the sense that all other programs that implement the specification simulate it.

Notice that the co-unit of the adjunction $\square_p:\text{synt}(\text{spec}(P)) \rightarrow P$ is not necessarily an isomorphism because **spec**(P) may not be powerful enough to fully characterise P (we cannot guarantee that the specification domain is expressive enough to capture the semantics of P in full). Hence, we are not even in the presence of an equivalence.

The direction of the co-unit reflects the fact that if we synthesise from the strongest specification of a program P , we obtain a program that cannot be stronger than P . Hence, the morphism \square_p provides a sort of "universal adaptor" between the program synthesised from **spec**(P) and P itself.



Although weaker than the existence of an inverse or a pseudo-inverse, the existence of a left adjoint to the functor **spec**: **PROG** \rightarrow **SPEC** is quite a strong property. This is not surprising because the ability to synthesise any specification is itself, in intuitive terms, a very strong property. In the literature, examples of synthesis of finite state automata from temporal logic specifications can be found, both from propositional linear temporal logic as above [85] and from branching time logic [30]. However, their generalisation to a full systems view is difficult. We shall in section **Error! Reference source not found.** that we can go a longer way in the context of formalisms that separate "Coordination" from "Computation".

Another important property that results from the properties of reflective functors is the following:

7.3.8 PROPOSITION – adjunctions and reflections

1. Every reflective functor defines an adjunction (in which it plays the role of right adjoint).
2. In every adjunction $\mathcal{C} \xrightleftharpoons[\mathcal{Q}]{\mathcal{P}} \mathcal{D}$, \mathcal{P} is reflective with reflector \mathcal{Q} and \mathcal{Q} is co-reflective with co-reflector \mathcal{P} .

This result allows us to derive from 7.2.9 two interesting cases of adjunctions:

7.3.9 COROLLARY

1. Let $\mathcal{C} : \mathbf{C} \rightarrow \mathbf{SET}$ be a functor. The functor $\mathcal{C} : \mathbf{spa}(\mathcal{C}) \rightarrow \mathbf{C}$ has for left adjoint the functor that maps each \mathbf{C} -object c to $\langle c, \emptyset \rangle$, and for right adjoint the functor that maps each \mathbf{C} -object c to $\langle c, \mathcal{C}(c) \rangle$.
2. The functor **alph** that maps **PROC** to $\mathbf{SET}_{\mathcal{A}}$ by forgetting behaviours has both a left and right adjoint. The left adjoint maps each alphabet $A_{\mathcal{A}}$ to the process $\langle A_{\mathcal{A}}, \emptyset \rangle$ and the right adjoint maps it to $\langle A_{\mathcal{A}}, \mathbf{tra}(A_{\mathcal{A}}) \rangle$.

7.3.10 COROLLARY

In any π -institution, the functor **sign:THEO** \rightarrow **SIGN** that maps theories to their underlying signatures has both a left and right adjoint. The left adjoint maps each signature to the theory $\langle _, c_{\mathcal{C}}(\emptyset) \rangle$ and the right adjoint maps it to $\langle _, \mathbf{gram}(_) \rangle$.

Basically, both results tell us how to map back and forth between processes and alphabets, and between theories and signatures,

7.3.11 EXERCISE

Workout direct proofs for both 7.3.9 and 7.3.10, and interpret the meaning of the natural transformations. Check how far 7.3.10 extends to presentations and strict presentations.

From 7.2.10 and 7.3.8 we get another obvious class of adjunctions:

7.3.12 COROLLARY

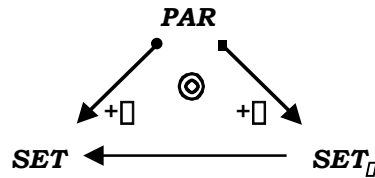
1. Every reflective subcategory defines an adjunction in which the inclusion functor is the right adjoint.
2. Every co-reflective subcategory defines an adjunction in which the inclusion is the left adjoint.

7.3.13 EXAMPLE – adjunctions between SET, PAR and SET₊

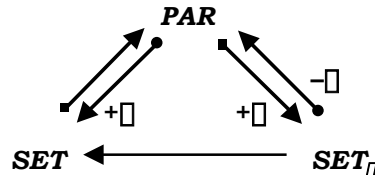
The fact that, as seen in 3.3.11, **SET** is a co-reflective subcategory of **PAR** tells us that the inclusion has a right-adjoint. As also seen in 3.3.11, this right adjoint (call it **+()**)

is the one that performs the traditional "elevation" of partial into total functions by extending sets with an "undefined" element, or "bottom", that serves as image for the elements in which the partial functions are undefined. This construction may well remind the reader of one of the functors over which the equivalence between **PAR** and **SET_⊥** was built in 7.1.12: the one that bears the same name. That is to say, we have the same kind of construction – the "elevation" – being performed over two different categories. Are they related?

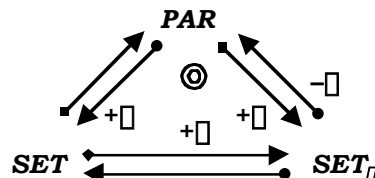
There is a "natural" way in which every pointed set can be viewed as a (normal) set: just forget the "added structure", i.e. the fact that it has a designated element. Note that this does not mean "through away the designated element", which is what the functor $-⊥$ (the pseudo-inverse of $+⊥$ in the equivalence) does. Going back to 3.2.1, we are mapping pointed sets $\langle A, \perp_A \rangle$ to the underlying set A and morphisms between two pointed sets $f: \langle A, \perp_A \rangle \rightarrow \langle B, \perp_B \rangle$ to the corresponding total function $f: A \rightarrow B$. This mapping defines **SET_⊥** as a concrete category over **SET**. The two elevations of **PAR** are related by this forgetful functor: the elevation to **SET** is simply the result of forgetting that there is a designated element in the elevation to **SET_⊥**, which is captured by the following commutative diagram:



Notice that the elevation to **SET_⊥** is explicitly recorded into a structure that is added to sets whereas the elevation to **SET** is merely a representation or encoding. This difference is well captured in the fact that it gives rise to an equivalence in the first case but "just" a reflective functor in the second. The elevation from **PAR** to **SET_⊥** is also reflective but the fact that it is a co-reflection is more interesting for the following reason: if we complete the diagram with the adjunctions that we have already built



we can see that the functor that forgets the designated elements admits a right adjoint that, again, performs another kind of elevation, this time superposing a designated element to every set. This is just the elevation of partial functions being performed on total ones as a particular case.



Notice that we obtain a commutative diagram of adjunctions (i.e. of reflective and co-reflective functors), but not of the functors in general: for instance, the diagram of $+⊥$ is, clearly, not a commutative one!

It is also important to point out, in the sequel of the remarks made in 7.1.12, that because **PAR** and **SET_□** are equivalent, we tend to look at them as being "the same", but they may bear quite different relationships to other categories like, in this case, **SET**: for instance, the "elevations" go in opposite directions, one from **PAR**, the other into **SET_□**; one is reflective and the other co-reflective. What is more interesting is that these are "technical" differences: conceptually, both **PAR** and **SET_□** provide a co-reflective representation for "normal" sets; the representations are different because, in spite of being equivalent, the two categories offer different structures and, hence, require different encodings of what is, essentially, the same kind of relationship.

This example also shows how diagrams of adjunctions can be useful to understand how different domains relate to each other; they provide a kind of "roadmap" or "classification scheme" that is essential to navigate among the different structures that one tends to find in the literature. For instance, we can enrich the previous diagram of adjunctions with the one that we obtained in 7.3.6:

$$\begin{array}{ccccc} \mathbf{SET} & \begin{array}{c} \xrightarrow{\quad} \\ \xleftarrow{\quad} \end{array} & \mathbf{PAR} & \begin{array}{c} +\square \\ \hline -\square \end{array} & \mathbf{SET}_{\square} & \begin{array}{c} \xrightarrow{2^{-}} \\ \xleftarrow{2^{-}} \end{array} & \mathbf{SET}^{\text{op}} \end{array}$$

The composition of these adjunctions gives us the well known adjunction between **SET** and **SET^{op}** performed by the powerset functor.

This kind of roadmap was used in [98,112] for formalising relationships between models of concurrency like transition systems, synchronisation trees, event structures, etc, in what constitutes one of the most striking examples of the expressive power of adjunctions. Each such model is endowed with a notion of morphism that captures a form of simulation as a behaviour-preserving mapping. Typical operations of process calculi are captured as universal constructions as exemplified in 4.3.8 for **PROC**. Reflections and co-reflections¹¹ are used for expressing the way one model is embedded in another: one of the functors in the adjunction embeds the more abstract model in the other, while the other functor abstracts away from some aspect of the representation.

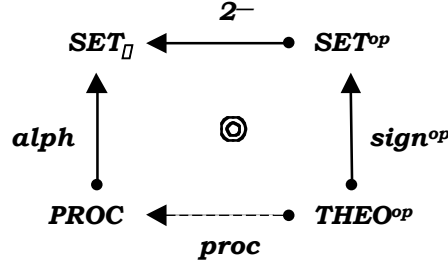
Instead of reproducing an example of such uses of adjunctions, which on its own would hardly capture the richness of the classification that is developed for different kinds of concurrency models in [98,112], we present a related kind of application: a duality between process models and specifications. More precisely, a duality between **PROC** and linear temporal logic specifications given by **THEO_{LTL}** as presented in [32] to show how both semantics domains – theories and models – can be made part of the same roadmap.

7.3.14 EXAMPLE – duality between processes and specifications

We start by recalling that **PROC** is concrete over **SET_□** through the forgetful functor **alph** and that **THEO** is concrete over **SET** through the forgetful functor **sign**. In fact, we proved in 7.3.9 and 7.3.10 that these functors are both reflective and co-reflective. Moreover, we have shown in 7.3.6 that the contravariant powerset functor **2⁻:SET^{op}→SET_□** is reflective. We are now going to show that **2⁻** can be lifted to **proc**:

¹¹ Further terminological confusion arises with respect to [112] where a co-reflection is an adjunction in which the reflective functor (the adjoint) is a full embedding, i.e. the straight generalisation of a full reflective subcategory.

THEO^{op} \square **PROC** as a reflective functor that makes the diagram (of reflective functors) below commute.



Let $\langle \square, \square \rangle$ be a theory. The required commutativity of the diagram fixes the choice of the alphabet for **proc**($\langle \square, \square \rangle$): the powerset 2^\square considered as a pointed set. We are going to choose for its behaviour the set $\{\square\square(2^\square)^\square \mid \square \models \square\}$, i.e. the least deterministic process that satisfies the properties given by the theory. It is not difficult to prove that we do obtain a functor.

If we consider now a process $\langle A_\square, \square \rangle$, the category $\langle A_\square, \square \rangle \square \mathbf{proc}$ consists of the specifications that are validated by the behaviours in \square after a suitable translation (what in 6.5.9 we called generalised models). This category has an initial object: the set of all sentences in the language of 2^A that are validated by the behaviours of \square translated by the unit of the powerset adjunction. It is the largest, not the smallest, because we are working with a contravariant functor. Hence, **proc** is indeed reflective, its reflector assigning to $\langle A_\square, \square \rangle$ the theory $\langle 2^A, \{p \mid \square; \square_A \models p\} \rangle$.

Notice that because reflective functors preserve limits (7.2.8), colimits of specifications are mapped by **proc** to limits of the corresponding processes. This is again a form of *compositionality*. It says that composition of specifications as given by colimits of configuration diagrams captures parallel composition of the corresponding processes taking into account the interactions given by the diagram.

The reader is encouraged to consult [32,33] for a wider discussion of the relationships between these two categories, namely in the context of what are called categorical institutions in [87].

7.3.15 EXERCISE

1. Workout the full proof of 7.3.14.
2. Relate this result to 6.5.18.
3. Since, in the diagram of 7.3.14, **sign^{op}** and **2⁻** are reflective and **alph** is co-reflective, why didn't we take the composition of **sign^{op}** with **2⁻** and the co-reflector of **alph** to obtain an adjunction from **PROC** to **THEO^{op}**?
4. What kind of generalisation into institutions can we hope for?

We would now like to extend this ability of adjunctions to relate different semantic models of concurrency, to different specification formalisms as captured by the categories of theories defined by institutions. For that purpose, we present in the next section a summary of the results published in [5].

7.4 Adjunctions in institutions

We start with an example in order to motivate the structures that are involved in mapping between specification formalisms as captured by institutions.

The typical temporal logics that have been used for the specification of reactive systems are based on linear time, of which the one we have been working with is only an example. However, sometimes, a branching time logic may be more justified. For instance, it is well known that verification techniques over branching time logic can be more effective. The expressive power of branching time logic can also be useful, especially in relation to progress properties related to required non-determinism, like responsiveness. So, we would like to have ways of mapping between specifications in these different logics that support translations back and forth between them, through which one can take advantage of the best features of each.

As an example, consider the specification of a "user-friendly" vending machine that, once it accepts a coin, will make a cake and a cigar available. Notice that this is not a property of the vending machine as specified in 3.5.6; the specification given therein allows behaviours in which, for instance, after a coin is accepted, no cakes and no cigars are delivered! One could think that adding a property like $(\text{coin} \mathbf{F}(\text{cake} \text{ cigar}))$ would solve the problem but it is easy to see that this requirement does not capture the availability of a cake or a cigar for the customer to choose; it requires that, in every behaviour, the acceptance of a coin is followed by the delivery of a cigar or a cake. Hence, it admits as an implementation a machine that only delivers cigars! Moreover, it forces the customer to take either the cake or the cigar among the options that are given, which does not make sense when that activity is not initiated by the machine. All this is because the trace-based semantics is not expressive enough to model choice; for that purpose, branching structures are required.

A logic in which such properties can be easily expressed is the branching time logic CTL^* . This logic is said to be branching because operators are provided that quantify over the possible future behaviours from the current state.

7.4.1 DEFINITION – CTL^* as an institution

The branching temporal logic institution CTL^* is defined as follows:

- Its category of signatures is **SET**.
- We define two classes of propositions (the state propositions \square_s and the path propositions \square_p) for a temporal signature \square :

$$\begin{aligned}\square_s \square &::= \square a \square \mid \square (\square \square_s) \square \mid \square (\square_s \square_s) \square \mid \square (\mathbf{A} \square_p) \\ \square_p \square &::= \square \square_s \square \mid \square (\square \square_p) \square \mid \square (\square_p \square_p) \square \mid \square (\square_p \mathbf{U} \square_p)\end{aligned}$$

The set of branching temporal propositions $\mathbf{gram}_{CTL^*}(\square)$ is the set of state propositions.

A signature morphism $f: \square \rightarrow \square'$ induces the translation $f: \mathbf{gram}_{CTL^*}(\square) \rightarrow \mathbf{gram}_{CTL^*}(\square')$ defined as follows:

$$f(\square_s) \square::= \square f(a) \square \mid \square \square f(\square_s) \square \mid \square f(\square_s) \square f(\square_s) \square \mid \square \mathbf{A} f(\square_p)$$

$$f(\Box_p) \Box \Box ::= \Box \Box f(\Box_s) \Box \Box \mid \Box \Box \Box f(\Box_p) \Box \Box \mid \Box \Box f(\Box_p) \Box f(\Box_p) \Box \Box \mid \Box f(\Box_p) \mathbf{U} f(\Box_p).$$

- The model functor \mathbf{mod}_{CTL^*} is defined as follows:
 - For every signature \Box , a branching \Box -model is a triple $\langle W, R, V \rangle$ with R a total relation on W and $V: \Box \Box 2^W$.
 - Given a branching model $M = \langle W, R, V \rangle$ we denote by $L(M)$ the set of all infinite sequences $\Box \Box 2^\Box$ of the form $\Box; V^1$ where $\Box: \Box \Box W$ is such that $\Box(i)R\Box(i+1)$ for every $i \Box$. We denote by $L(M, s)$ the subset of $L(M)$ that is generated by the sequences $\Box_s: \Box \Box W$ such that $\Box_s(0) = s$, i.e. $L(M, s)$ contains the subset of paths that begin at state s .
 - Let $M_1 = \langle W_1, R_1, V_1 \rangle$ and $M_2 = \langle W_2, R_2, V_2 \rangle$ be \Box -models. A (p-)morphism from M_1 to M_2 is a map $f: W_1 \Box W_2$ such that:
 - (a) $sR_1 t$ implies $f(s)R_2 f(t)$,
 - (b) $f(s)R_2 u$ implies the existence of $t \Box W_1$ such that $sR_1 t$ and $f(t) = u$,
 - (c) $s \Box V_1(a)$ iff $f(s) \Box V_2(a)$.
 - Let $f: \Box_1 \Box \Box_2$ be a signature morphism. If $\langle W_2, R_2, V_2 \rangle$ is a \Box_2 -model, then $\langle W_2, R_2, f; V_2 \rangle$ is a \Box_1 -model called the *f-reduct* of $\langle W_2, R_2, V_2 \rangle$.
- The satisfaction relation is defined as follows: the truth of a \Box -proposition \Box in $M = \langle W, R, V \rangle$ at state $s \Box W$ (which we write $(M, s) \models_\Box \Box$) is inductively defined as for LTL except for the operator \mathbf{A} for which
 - $(M, s) \models_\Box \mathbf{A} \Box$ iff, for every $\Box_s \Box L(M, s)$, $\Box_s \models_\Box^0 \Box$.

The branching temporal proposition \Box is said to be true in M , which we denote by $M \models_\Box \Box$, iff $(M, s) \models_\Box \Box$ at every state s of W .

Notice how the new operator, \mathbf{A} , quantifies over all possible paths that start from the current state.

The reader is invited to check that the satisfaction condition holds, i.e. prove the following property:

7.4.2 PROPOSITION – satisfaction condition

Let $f: \Box_1 \Box \Box_2$ be a signature morphism. For every $M \Box |\mathbf{mod}_{CTL^*}(\Box_2)|$ and $\Box \Box \mathbf{gram}_{CTL^*}(\Box_1)$, $M \models_{\Box_2} f(\Box)$ iff $\mathbf{mod}_{CTL^*}(f)(m) \models_\Box \Box$.

7.4.3 COROLLARY

CTL^* as defined in 7.4.1 is an institution.

As an example of a specification in CTL^* , consider the user-friendly vending machine

```

specification user-friendly vending machine is
signature      coin, cake, cigar
axioms        beg    $\mathbf{A}(\neg \text{cake} \Box \neg \text{cigar}) \Box \mathbf{A}(\text{coin} \Box (\neg \text{cake} \Box \neg \text{cigar}) \mathbf{W} \text{coin})$ 
                  coin   $\mathbf{A}((\neg \text{coin}) \mathbf{W} (\text{cake} \Box \text{cigar}))$ 
                  coin   $(\mathbf{EX} \text{cake} \Box \mathbf{EX} \text{cigar})$ 
                  (cake cigar)  $\mathbf{A}(\neg \text{cake} \Box \neg \text{cigar}) \mathbf{W} \text{coin}$ 
                  cake    $(\neg \text{cigar})$ 
    
```

The operator **E** is the dual of **A**: it expresses the existence of a path from the current state in which the given property holds. Notice the use of the conjunction in *(coin (EXcake \wedge EXcigar))*; it requires the machine to give the customer the choice; hence, for instance, if the machine runs out of cakes, it may not accept coins even if cigars are still available.

It is clear from the definition of CTL^* that this logic "incorporates" LTL in the sense that it can express at least as much as LTL . A theory in LTL expresses properties about all possible behaviours of a system taken as infinite sequences of states. In CTL^* , this quantification can be made explicit through the operator **A**. Hence, it should be straightforward to map a theory of LTL to a theory of CTL^* by qualifying every linear proposition with **A**.

This syntactic transformation between the two languages respects the translations defined by signature morphisms, i.e. is "natural" on signatures. Indeed, it is captured by a natural transformation:

7.4.4 DEFINITION/PROPOSITION

The family of functions $\llbracket \cdot \rrbracket : \mathbf{gram}_{LTL}(\Sigma) \rightarrow \mathbf{gram}_{CTL^*}(\Sigma)$ defined by $\llbracket A \rrbracket = \mathbf{A}A$ is a natural transformation from \mathbf{gram}_{LTL} to \mathbf{gram}_{CTL^*} .

There is also a way in which this translation can be claimed to be "correct". Every branching structure gives rise to a linear one in a natural way:

7.4.5 DEFINITION/PROPOSITION

Let $\llbracket \cdot \rrbracket$ map every branching model $M = \langle W, R, V \rangle$ to the linear model $L(M)$ defined in 7.4.1. Given any branching Σ -models $M = \langle W, R, V \rangle$ and $M' = \langle W', R', V' \rangle$, and a p-morphism $f: M \rightarrow M'$, let $\llbracket f \rrbracket$ be the inclusion $L(M) \rightarrow L(M')$ that is induced by the properties of the morphism. The mapping $\llbracket \cdot \rrbracket$ is a functor $\mathbf{mod}_{CTL^*}(\Sigma) \rightarrow \mathbf{mod}_{LTL}(\Sigma)$. The family $\langle \llbracket \cdot \rrbracket \rangle_{\Sigma \in \mathbf{SIGN}}$ defines a natural transformation $\mathbf{mod}_{CTL^*} \rightarrow \mathbf{mod}_{LTL}$.

That is to say, we generate from every branching structure M the linear structure $L(M)$ that consists of all possible paths (runs) through M .

The syntactic and semantic transformations agree in the following sense:

7.4.6 DEFINITION/PROPOSITION

If $M = \langle W, R, V \rangle$ is a branching Σ -model and $\varphi \in \mathbf{gram}_{LTL}(\Sigma)$, then $M \models \varphi$ iff $\llbracket M \rrbracket \models \llbracket \varphi \rrbracket$.

This relationship between the syntax and semantics of the given institutions allows us to define the intended functor between the corresponding categories of theories:

7.4.7 DEFINITION/PROPOSITION

The mapping $\mathcal{T} : \mathbf{THEO}_{LTL} \rightarrow \mathbf{THEO}_{CTL^*}$ defined by $\mathcal{T}(\langle \Sigma, \varphi \rangle) = \langle \Sigma, c(\llbracket \varphi \rrbracket) \rangle$ is a functor.

By $\mathbf{A}\varphi$ we are denoting the set $\{A\varphi \mid \varphi \in \mathbf{gram}_{LTL}(\Sigma)\}$ and by c the closure operator of CTL^* .

This functor allows us to translate any specification (theory) in LTL to a specification in CTL^* . This translation is "canonical" in the following sense:

7.4.8 PROPOSITION

The functor $\mathcal{T}:\mathbf{THEO}_{LTL} \rightarrow \mathbf{THEO}_{CTL^*}$ is co-reflective (has a right adjoint).

It is not difficult to "guess" the nature of the right adjoint. Because \mathcal{T} computes direct images through \square , its right adjoint computes inverse images, i.e. the adjunction is given by a generalisation to closure operators of the well known Galois connection between direct and inverse images of sets. The unit of the adjunction is given by the inclusion $\square\square\square_{\square}^{-1}(c(\square_{\square}(\square)))$.

7.4.9 PROPOSITION

The mapping $\mathcal{V}:\mathbf{THEO}_{CTL^*} \rightarrow \mathbf{THEO}_{LTL}$ given by $\mathcal{V}(\langle \square, \square \rangle) = \langle \square, \square_{\square}^{-1}(\square) \rangle$ is a co-reflector (right adjoint) of \mathcal{T} .

PROOF (of 7.4.7 and 7.4.8)

The functor \mathcal{V} is a co-reflector of \mathcal{T} iff for every theory $\langle \square, \square \rangle$ of LTL, the pair $(\langle \square, \square_{\square}^{-1}(c(\square_{\square}(\square))) \rangle, id_{\square})$ is a reflection. Consider $f:\langle \square, \square \rangle \rightarrow \langle \square', \square'^{-1}(\square') \rangle$. We have to prove that there is a unique CTL^* morphism $f':\langle \square, c(\square_{\square}(\square)) \rangle \rightarrow \langle \square', \square' \rangle$ such that $id_{\square'} \circ f' = f$. Unicity is automatically guaranteed by this equation. All that remains is the proof that f is a theory morphism $\langle \square, c(\square_{\square}(\square)) \rangle \rightarrow \langle \square', \square' \rangle$ in CTL^* . Let $\square_{\square}(\square) \vdash_{\square} \square$. We have to prove that $f(\square)\square'$. Because $\square_{\square}(\square) \vdash_{\square} \square$ we have $f(\square_{\square}(\square)) \vdash_{\square'} f(\square)$ (a consequence of the satisfaction condition of CTL^*). But $f(\square_{\square}(\square)) = \square_{\square'}(f(\square))$ because \square is a natural transformation. Hence, $f(\square)\square_{\square'}c(f(\square))$. On the other hand, $f(\square)\square_{\square'}^{-1}(\square')$ because f was taken as a theory morphism $\langle \square, \square \rangle \rightarrow \langle \square', \square'^{-1}(\square') \rangle$. Hence, $f(\square)\square'$.

The co-reflector "forgets" the branching nature of time in CTL^* by retaining only those propositions \square for which $\mathbf{A}\square$ is a theorem in CTL^* , i.e. it retains those truths that hold for every possible path.

The existence of the adjunction means that, in order to prove that a CTL^* -theory BT provides an interpretation (refinement) of an LTL -theory LT , it is equivalent to prove $LT \sqsubseteq \mathcal{V}(BT)$ or $\mathcal{T}(LT) \sqsubseteq BT$. For practical purposes, the inclusion $\mathcal{T}(LT) \sqsubseteq BT$ is easier to prove because it can be lifted to presentations. Indeed, if we take the category \mathbf{PRES}_{LTL} of the theory presentations of LTL , the adjunction between presentations and theories (3.6.4) allows us to extend the adjunction between \mathbf{THEO}_{CTL^*} and \mathbf{THEO}_{LTL} to one between \mathbf{THEO}_{CTL^*} and \mathbf{PRES}_{LTL} . Hence, the inclusion $\mathcal{T}(LT) \sqsubseteq BT$ can be proved at the level of a presentation of LT . The converse, however, does not hold because, although there is also an adjunction between \mathbf{THEO}_{CTL^*} and \mathbf{PRES}_{CTL^*} , the right adjoint does not go in the same direction as \mathcal{V} .

The actual relationship between \mathbf{THEO}_{CTL^*} and \mathbf{THEO}_{LTL} is stronger than what we proved. The proof above showed us that every LTL -theory $\langle \square, \square \rangle$ is included in $\langle \square, \square_{\square}^{-1}(c(\square_{\square}(\square))) \rangle$ but, in fact, they are equal. That is, when translated back from its image in CTL^* , an LTL -theory does not gain any theorems. This result can be proved by noticing that every linear structure can be generated by a branching one, i.e. the natural transformation \square consists of surjective mappings, which gives us the "faithfulness" of the right-adjoint (7.2.11). We will generalise this result below but it is important to realise that this means that the translation from LTL to CTL^* is "conservative", i.e. the representation of LTL in CTL^* is faithful.

Notice that, in the proof above, no use was made of the syntactic transformation itself. Only the fact that \square is a natural transformation was used, which indicates that the relationship between LTL and CTL^* can be generalised to other institutions.

In order to perform the generalisation, let us first analyse what in the example above can be cast directly in categorical terms. The basic ingredients in our example were:

- a natural transformation $\square: \mathbf{gram}_{LTL} \rightarrow \mathbf{gram}_{CTL^*}$;
- a natural transformation $\square: \mathbf{mod}_{CTL^*} \rightarrow \mathbf{mod}_{LTL}$;
- the invariance condition $M \models_{\square(\square)} \square \text{ iff } \square(M) \models_{\square} \square$.

These are exactly the ingredients found in institution morphisms [61] and institution maps [87].

7.4.10 DEFINITION – institution morphism

Let $\square = \langle \mathbf{SIGN}, \mathbf{gram}, \mathbf{mod}, \models \rangle$ and $\square' = \langle \mathbf{SIGN}', \mathbf{gram}', \mathbf{mod}', \models' \rangle$ be institutions. An *institution morphism* $\square: \square \rightarrow \square'$ is a triple $\langle \square, \square, \square \rangle$ where:

- $\square: \mathbf{SIGN} \rightarrow \mathbf{SIGN}'$ is a functor;
- $\square: \square; \mathbf{gram} \rightarrow \mathbf{gram}'$ is a natural transformation;
- $\square: \square; \mathbf{mod} \rightarrow \mathbf{mod}'$ is a natural transformation

such that the following property (the *invariance condition*) holds for any signature $\square \in \mathbf{SIGN}$, $m \in \mathbf{mod}(\square)$ and $\square \in \mathbf{gram}(\square(\square))$: $m \models_{\square(\square)} \square \text{ iff } \square(m) \models'_{\square(\square)} \square$.

7.4.11 DEFINITION – institution map

Let $\square = \langle \mathbf{SIGN}, \mathbf{gram}, \mathbf{mod}, \models \rangle$ and $\square' = \langle \mathbf{SIGN}', \mathbf{gram}', \mathbf{mod}', \models' \rangle$ be institutions. An *institution map* $\square: \square \rightarrow \square'$ is a triple $\langle \square, \square, \square \rangle$ where:

- $\square: \mathbf{SIGN} \rightarrow \mathbf{SIGN}'$ is a functor;
- $\square: \mathbf{gram} \rightarrow \square; \mathbf{gram}'$ is a natural transformation;
- $\square: \square; \mathbf{mod}' \rightarrow \mathbf{mod}$ is a natural transformation

such that the following property (the *invariance condition*) holds for any signature $\square \in \mathbf{SIGN}$, $m \in \mathbf{mod}(\square(\square))$ and $\square \in \mathbf{gram}(\square): \square(m') \models_{\square} \square \text{ iff } m \models'_{\square(\square)} \square(\square)$.

7.4.12 PROPOSITION

Through 7.4.4, 7.4.5 and 7.4.6 we have defined both a map $LTL \rightarrow CTL^*$ and a morphism $CTL^* \rightarrow LTL$.

Indeed, the fact that the relationship between the two institutions is based on the identity functor between their categories of signatures blurs the difference between both concepts (morphism and map).

The existence of the two functors \mathcal{T} and \mathcal{U} between \mathbf{THEO}_{LTL} and \mathbf{THEO}_{CTL^*} is also a consequence of the existence of a map and a morphism between the two institutions:

7.4.13 PROPOSITION

Let $\square = \langle \square, \square, \square \rangle: \square \rightarrow \square$ be an institution map. The functor \square can be extended to a functor $\mathbf{THEO}_{\square} \rightarrow \mathbf{THEO}_{\square}$ by establishing $\square(\langle \square, \square \rangle) = \langle \square(\square), c(\square(\square)) \rangle$.

7.4.14 PROPOSITION

Let $\square = \langle \square, \square', \square' \rangle: \square \rightarrow \square$ be an institution morphism. The functor \square can be extended to a functor $\mathbf{THEO}_{\square} \rightarrow \mathbf{THEO}_{\square}$ by establishing $\square(\langle \square', \square' \rangle) = \langle \square(\square'), \square'^{-1}(\square') \rangle$.

We can now generalise the results on the adjunction between the categories of theories of two institutions:

7.4.15 PROPOSITION

Let $\mathbb{I} = \langle \text{SIGN}, \text{gram}, \text{mod}, \models \rangle$ and $\mathbb{I}' = \langle \text{SIGN}', \text{gram}', \text{mod}', \models' \rangle$ be institutions, $\mathbb{I} = \langle \mathbb{I}, \mathbb{I}, \mathbb{I} \rangle: \mathbb{I} \rightarrow \mathbb{I}$ an institution map and $\langle \mathbb{I}, \mathbb{I}', \mathbb{I}' \rangle: \mathbb{I} \rightarrow \mathbb{I}'$ a morphism such that \mathbb{I} is a right adjoint of \mathbb{I}' , and, for every $\mathbb{I} \models \text{SIGN}$, $\mathbb{I}_{\mathbb{I}} = \text{gram}(\mathbb{I}_{\mathbb{I}}); \mathbb{I}'_{\mathbb{I}(\mathbb{I})}$ where \mathbb{I} is the unit of the adjunction. Then,

1. The functor $\mathcal{U}: \text{THEO}_{\mathbb{I}} \rightarrow \text{THEO}_{\mathbb{I}'}$ induced by the morphism $\langle \mathbb{I}, \mathbb{I}', \mathbb{I}' \rangle$, is a right adjoint of the functor $\text{THEO}_{\mathbb{I}} \rightarrow \text{THEO}_{\mathbb{I}'}$ induced by the map $\langle \mathbb{I}, \mathbb{I}, \mathbb{I} \rangle$.
2. If each component of \mathbb{I} is surjective, i.e. if the institution morphism is sound in the sense of [61], then the units $\mathbb{I}_{\mathbb{I}}$, as theory morphisms, are conservative.

PROOF:

this is a direct generalisation of the proof of 7.4.9.

That is to say, adjunctions on signatures can be lifted to adjunctions of theories provided that the left adjoint is associated with a map and the right adjoint with a morphism of institutions. A compatibility result is required, $\mathbb{I}_{\mathbb{I}} = \text{gram}(\mathbb{I}_{\mathbb{I}}); \mathbb{I}'_{\mathbb{I}(\mathbb{I})}$, to make sure that both the map and the morphism make, essentially, the same translations. Notice that the invariance condition relating \mathbb{I} and \mathbb{I}' automatically generates a similar property for \mathbb{I} . The result on "conservative" representations of one formalism into another is also important: basically, it says that no new theorems arise when a theory is translated from one formalism to another.

This result shows that there is a very strong relationship between institution morphisms and maps, as suggested by the fact that they make use of essentially the same transformations between languages and models. The difference between them, which is evident in the directions taken by the transformations vis-à-vis the functor between the categories of signatures, can be explained more easily when we see that they correspond to the two directions of an adjunction. Notice that the map takes the direction of the left adjoint while the morphism takes the direction of the right adjoint. These directions are very much consistent with the accepted view of maps as providing representations and morphisms projections of one institution into another.

In fact, the result above is more general in that the existence of a map (resp. morphism) and a right (resp. left) adjoint for the signature functor guarantees the existence of a morphism (resp. map) in the other direction that generates a right (resp. left) adjoint to the theory functor:

7.4.16 PROPOSITION

Let $\mathbb{I} = \langle \text{SIGN}, \text{gram}, \text{mod}, \models \rangle$ and $\mathbb{I}' = \langle \text{SIGN}', \text{gram}', \text{mod}', \models' \rangle$ be institutions,

1. if $\mathbb{I} = \langle \mathbb{I}, \mathbb{I}, \mathbb{I} \rangle: \mathbb{I} \rightarrow \mathbb{I}$ is a map such that the functor \mathbb{I} has a right adjoint \mathbb{I} , then
 - a) the triple $\langle \mathbb{I}, \mathbb{I}', \mathbb{I}' \rangle$ where \mathbb{I}' is the natural transformation defined by $\mathbb{I}'_{\mathbb{I}} = \mathbb{I}_{\mathbb{I}(\mathbb{I})}; \text{gram}'(\mathbb{I}_{\mathbb{I}})$ and \mathbb{I} is the natural transformation defined by $\mathbb{I}_{\mathbb{I}} = \text{mod}'(\mathbb{I}_{\mathbb{I}}); \mathbb{I}'_{\mathbb{I}(\mathbb{I})}$, is an institution morphism $\mathbb{I} \rightarrow \mathbb{I}'$
 - b) the functor $\mathcal{T}: \text{THEO}_{\mathbb{I}} \rightarrow \text{THEO}_{\mathbb{I}'}$ induced by the map has a right adjoint – the functor $\mathcal{U}: \text{THEO}_{\mathbb{I}'} \rightarrow \text{THEO}_{\mathbb{I}}$ induced by the morphism, i.e. $\mathcal{U}(\langle \mathbb{I}', \mathbb{I}' \rangle) = \langle \mathbb{I}(\mathbb{I}'), \mathbb{I}_{\mathbb{I}(\mathbb{I}')}^{-1}(\mathbb{I}'^{-1}(\mathbb{I}')) \rangle$.

2. if $\langle \mathcal{I}, \mathcal{I}', \mathcal{I} \rangle : \mathcal{I} \rightarrow \mathcal{I}'$ is a morphism such that the functor \mathcal{I} has a left adjoint \mathcal{I}' , then
- the triple $\langle \mathcal{I}, \mathcal{I}', \mathcal{I} \rangle$ where \mathcal{I} is the natural transformation defined by $\mathcal{I}_{\mathcal{I}} = \mathbf{gram}(\mathcal{I}_{\mathcal{I}}; \mathcal{I}'_{\mathcal{I}(\mathcal{I})})$ and \mathcal{I}' is the natural transformation defined by $\mathcal{I}'_{\mathcal{I}(\mathcal{I})} = \mathbf{mod}(\mathcal{I}_{\mathcal{I}})$ is an institution map from $\mathcal{I} \rightarrow \mathcal{I}'$.
 - the functor $\mathcal{U} : \mathbf{THEO}_{\mathcal{I}} \rightarrow \mathbf{THEO}_{\mathcal{I}'}$ induced by the morphism has a left adjoint – the functor $\mathcal{T} : \mathbf{THEO}_{\mathcal{I}'} \rightarrow \mathbf{THEO}_{\mathcal{I}}$ induced by the map, $\mathcal{T}(\langle \mathcal{I}, \mathcal{I} \rangle) = \langle \mathcal{I}(\mathcal{I}), \mathcal{I}'_{\mathcal{I}(\mathcal{I})} \circ (\mathcal{I}_{\mathcal{I}}(\mathcal{I})) \rangle$. ■

That is to say, provided that there is an adjunction between the categories of signatures of two institutions, maps and morphisms between them can be defined, interchangeably, that provide adjunctions for the functor between the corresponding categories of theories.

7.5 Coordinated categories

In this last section of the last chapter of Part Two and, hence, what could have been the closing paragraphs of an Introduction to Category Theory for Software Scientists and Practitioners, we address one of the topics that have been closest to the hearts of the research team that has been working on CommUnity, i.e. the subject of part three: the formalisation of the separation of concerns that is known as "Coordination". This is both a justification for stopping here – the reader will not need any more categorical "ammunition" to attack part three – and having gone this far – the kind of application discussed in part three is intrinsically related to this topic and, even if a quicker route could have been taken, everybody knows that motorways are not the best ways for getting to know a region.

An introduction to this subject has already been given in section 5.2 as part of the motivation for studying the behaviour of functors in relation to universal constructions; the reader is invited to read it (once again) as well as, if possible, what I consider to be the best introduction to "Coordination": Arbab's gem "What Do You Mean, Coordination?" [3]. The central idea of this research area is to investigate the extent up to which a given formalism can separate between the mechanisms that coordinate the interactions that are responsible for emergent behaviour from the description of what in systems is responsible for the computations that ensure the functionalities of the services that individual system components provide.

For instance, object-oriented systems do not go a long way in supporting that separation. Because interactions in object-oriented approaches are based on *identities* [72], 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 the levels of agility required by the "just-in-time" binding mechanisms of (web) services; any change on 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, of the objects with which the new collaborations are established. That is to say, as beautifully put in [99], feature calling is, for interconnections, what assembly language represents for computations. On the contrary, interactions in a service-oriented approach should be based only on the description of what is required, thus decoupling the "what one wants to be done" from the "who does it". In the context of the "societal metaphor" that we have been using in the book, it is interesting to note that this shift from "object" to "service"-oriented inter-

actions mirrors what has been happening already in human society: more and more, business relationships are being established in terms of acquisition of services (e.g. 1000 Watts of lighting for your office) instead of products (10 lamps of 100 Watts each for the office).

Our introduction to section 5.2 has disclosed most of the "secrets" of the mathematical characterisation that we started to develop in [35] as a systematic study of the nature and properties of the separation between "Computation" and "Coordination" concerns. Now that the reader has more categorical background, we can revisit the motivation that has been already delivered. Notice that we shall systematically work with co-limits just to fix a direction of the "component-of" relationship and use it consistently. However, those that are more accustomed to limits can simply switch the direction of the arrow, i.e. work in the opposite category.

- we model this separation by a forgetful functor $\mathbf{int}:\mathbf{SYS}\rightarrow\mathbf{INT}$ where the category \mathbf{SYS} stands for the representations (models, behaviours, specifications, programs, ...) of the components out of which systems can be put together, and the category \mathbf{INT} captures the "interfaces" through which interconnections between system components can be established;
- \mathbf{int} should be faithful (5.1.7) so that morphisms in \mathbf{SYS} (the "component-of" relationship) do not induce more relationships between components 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. Using the terminology that we introduced in the previous chapter, \mathbf{SYS} is concrete over \mathbf{INT} .
- because we use diagrams for modelling configurations of complex systems and colimits to obtain emergent behaviour, \mathbf{int} should lift colimits (5.2.1): when we interconnect system components in a (configuration) diagram, 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 as given by the colimit of the original diagram. We have already mentioned that this property expresses (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. For instance, we saw in 6.1.22 that split fibre-(co)complete (co)fibrations lift limits.
- it is also clear that \mathbf{int} should preserve colimits (5.2.1): every interconnection of system components should be an interconnection of the underlying interfaces, i.e. computations should not make a configuration of system components "viable", in the sense that it admits a colimit, when the underlying configuration of interfaces is not. This is another form of the required "non-interference". Given that \mathbf{int} is faithful, this means that all colimits in \mathbf{SYS} are concrete (6.1.9).

Lifting and preservation of colimits 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} , a situation that we have already encountered for \mathbf{PROC} through the functor \mathbf{alph} and for the category of theories (or presentations) \mathbf{THEO} of any (π) -institutions through the functor \mathbf{sign} . In the case of processes, this means that the set of behaviours does not interfere with the interconnections; and in the case of theories, that interconnections are established just by name bindings.

Both examples allow us to illustrate another intuitive property of the separation that is not captured by those mentioned so far. Consider, for instance, processes. Taking pullbacks as the most basic form of interconnection, we can notice that the "middle" process through which we express the interconnection is "always" idle, i.e. has all possible behaviours. Indeed, the set of behaviours that is present in the middle process does not interfere neither in the interconnection, which is expressed at the level of the alphabets, nor in the calculation of the set of behaviours of the resulting process, which is defined through the intersection of the inverse images of the sets of behaviours of the other two component processes. Hence, there is a sort of "canonical" middle processes: those that are idle. Notice that their duals, the empty processes, do not make good middle processes because they do not admit any incoming morphisms...

The same happens with theories and theory presentations: the middle object in a pushout is "always" empty (or the closure of the empty set of axioms) because the theorems that result from the pushout are computed from the pushout of the signatures and the theorems of the other two components. This seems to be saying that the middle objects that we use for interconnecting components, be it for pullbacks or pushouts, are, essentially, interfaces, which makes all the sense from the point of view of the separation of "Coordination" from "Computation". How can we express this property in categorical terms?

Basically, and taking the colimit approach as exemplified by, for instance, theories, what we want is to be able to assign to every interface $C:\mathbf{INT}$ a component $s(C):\mathbf{SYS}$ such that, for every morphism $f:C \rightarrow \mathbf{int}(S)$, there is a morphism $g:s(C) \rightarrow S$ such that $\mathbf{int}(g)=f$. That is to say, we want every interface C to have a "realisation" as a system component $s(C)$ in the sense that, using C to interconnect a component S , which is achieved through a morphism $f:C \rightarrow \mathbf{int}(S)$, is tantamount to using $s(C)$ through any $g:s(C) \rightarrow S$ such that $\mathbf{int}(g)=f$. Notice that, because \mathbf{int} is faithful, there is only one such g , which means that f and g are, essentially, the same. That is, sources of morphisms in diagrams in \mathbf{SYS} are, essentially, interfaces. We would use the dual property to characterise what happens with processes.

Such a realisation is called a *discrete lift* in [1], and a functor \mathbf{int} for which every object $C:\mathbf{INT}$ admits a discrete lift is said to have *discrete structures*.

7.5.1 DEFINITION – discrete lifts/structures

Given a concrete category $\mathbf{C}:\mathbf{D} \rightarrow \mathbf{C}$, a *discrete lift* for $c:\mathbf{C}$ is a \mathbf{D} -object d such that $\mathbf{C}(d)=c$ and, for every morphism $f:c \rightarrow \mathbf{C}(d')$, there is a morphism $g:d \rightarrow d'$ such that $\mathbf{C}(g)=f$. The functor (concrete category) is said to have *discrete structures* whenever every \mathbf{C} -objects admits a discrete lift.

The dual notion is called *indiscrete lift* and the functor (concrete category) is said to have *indiscrete structures*.

When \mathbf{int} lifts and preserves colimits, this property allows us to replace every "middle" object in a configuration diagram by the discrete lift of the underlying interface: both diagrams will have the same colimits. For all "practical" purposes, this means that we can use more economical graphical representations for configuration diagrams by showing only the interfaces of the middle objects that interconnect components.

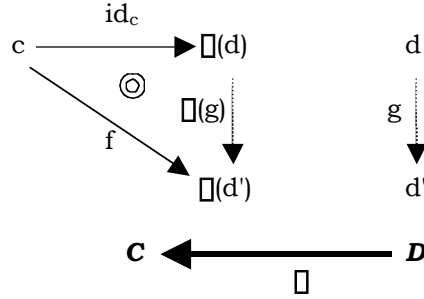
It is easy to see that the indiscrete lift of a process alphabet A_{\square} is $\langle A_{\square}, \mathbf{tra}(A_{\square}) \rangle$ and the discrete lift of a signature \square is the theory $\langle \square, c_{\square}(\emptyset) \rangle$. They also admit their dual versions – i.e. signatures have indiscrete lifts (inconsistent theories) and alphabets have discrete lifts (empty processes) – but these are not the ones that interest us for system configuration: they disable rather than enable interaction!

The more attentive reader is probably having a feeling of *déjà vu*... Indeed, these (in)discrete lifts are the objects involved in the (co)reflections that define the corresponding forgetful functors as (co)reflectors (see 7.3.9 and 7.3.10):

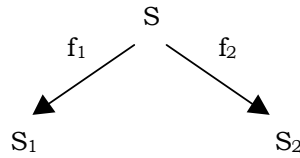
7.5.2 PROPOSITION

Every concrete category $\square : \mathbf{D} \rightarrow \mathbf{C}$ that has discrete structures is reflective, the reflections (i.e. the components of the unit) being identities.

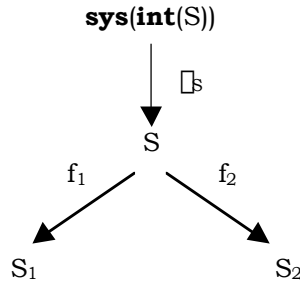
The proof of this result is immediate once one transcribes the definition of discrete lifts to diagrams:



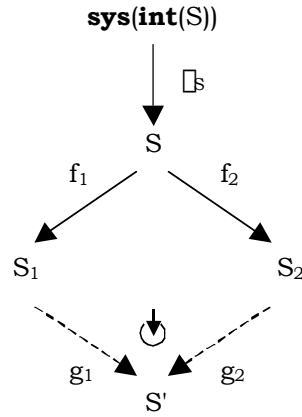
Notice that, \square being faithful, the co-reflections are epis (7.2.11). Actually, this is the property that allows us to replace the middle objects that perform interconnections by the discrete lifts of their underlying interfaces. Indeed, denoting by **sys** the reflector of **int**, every diagram



defines, by composition



Both diagrams admit the same pushouts because, \square being epi,



$\sqsubseteq_S; f_1; g_1 = \sqsubseteq_S; f_2; g_2$ implies $f_1; g_1 = f_2; g_2$.

7.5.3 EXERCISE

Prove that both diagrams have, indeed, the same pushouts.

He have now all the ingredients for our proposed characterisation of the formalisms that separate "Coordination" from "Computation":

7.5.4 DEFINITION – coordinated category

A concrete category (faithful functor) $\sqsubseteq: \mathbf{D} \rightarrow \mathbf{C}$, is said to be *coordinated* when:

- \sqsubseteq lifts colimits
- \sqsubseteq has discrete structures

In these circumstances, we also say that \mathbf{D} is coordinated over \mathbf{C} (via \sqsubseteq).

We have omitted the requirement on the preservation of colimits. This is because:

7.5.5 EXERCISE

Prove that coordinated functors preserve colimits.

As examples, we have already seen that theories and theory presentations of any (π) institution constitute a concrete category that is coordinated over their signatures, and that the (dual of) **PROC** is coordinated over (the dual of) alphabets. In Part Three of the book, we will see an example related to architectural description languages, the language CommUnity. We end this section with a "genuine" example: a simplified version of the language Gamma [11].

Before that, we would like to point out that the properties that characterise **SYS** as being coordinated over **INT** make **SYS** "almost" topological over **INT**. To be topological [1], **int** would have to lift colimits uniquely, which would make the concrete category amnesic (6.1.4). As far as the algebraic properties of the underlying formalism are concerned, this is not a problem because every concrete category can be modified

to produce an amnesic, concretely equivalent version. However, and although **PROC** is indeed amnesic, **PRES**, for instance, is not and neither is CommUnity. This is the "closest" characterisation we have to a "classical" mathematical structure: topological categories abound in Mathematics and other areas of Computer Science. In the areas related to Software Engineering, namely those in which one welcomes, or cannot avoid, "user intervention", one tends to work "up to isomorphism" more than "up to equality". In the case of the lifting of colimits, this means that there can be room for choosing between different, but isomorphic, system representations, for instance, alternative presentations of the same theory: one tends not to care whether a given conjunction ends up represented as $a \sqcap b$ or $b \sqcap a$.

We end this section with a brief discussion of an example borrowed from coordination formalisms: the language Gamma [11], which is based on the chemical reaction paradigm [16].

7.5.6 DEFINITION – Gamma programs

A Gamma program P consists of:

- a signature $\square = \langle S, \square, \sqcap \rangle$, where S is a set of sorts, \square is a set of operation symbols and \sqcap is a set of relation symbols, representing the data types that the program uses;
- a set of reactions, each of which is of the form:

$$R \quad \equiv \quad X, t_1, \dots, t_n \sqcap t'_1, \dots, t'_m \sqcap c$$

where

1. X is a set (of variables); each variable is typed by a data sort in S ;
2. $t_1, \dots, t_n \sqcap t'_1, \dots, t'_m$ is the action of the reaction – a pair of sets of terms over X ;
3. c is the reaction condition – a proposition over X .

An example of a Gamma program is the following producer of burgers and salads from, respectively, meat and vegetables:

```
PROD  $\equiv$ 
  sorts      meat, veg, burger, salad
  ops        vprod: veg  $\sqcap$  salad, mprod: meat  $\sqcap$  burger
  reactions  m:meat, m  $\sqcap$  mprod(m)
             v:veg, v  $\sqcap$  vprod(v)
```

The parallel composition of Gamma programs, as defined in [11], is a program consisting of all the reactions of the component programs. Its behaviour is obtained by executing the reactions of the component programs in any order, possibly in parallel. This leads us to the following notion of morphism.

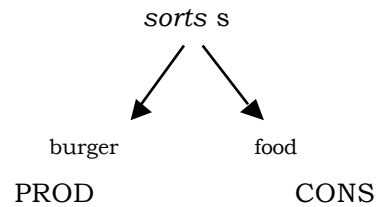
7.5.7 DEFINITION – morphisms of Gamma programs

A morphism \square between Gamma programs P_1 and P_2 is a morphism between the underlying data signatures such that $\square(P_1) \sqcap P_2$, i.e., P_2 has more reactions than P_1 .

In order to illustrate system configuration in Gamma, let us consider that we want to interconnect the producer with the following consumer:

CONS \models sorts food, waste
 ops cons: food \sqcup waste
 reactions f:food, f \sqcup cons(f)

The interconnection of the two programs is based on the identification of the food the consumer consumes, that is, the interconnection is established between their data types. For instance, the coordination of the producer and the consumer based on meat is given by the following interconnection:



Gamma is, indeed, coordinated over the category of data types:

- the forgetful functor dt from Gamma programs to data types is faithful;
- given any diagram in the category Gamma, a colimit $\coprod_i (dt(P_i) \sqcup \coprod_j R_j)$ of the corresponding diagram in the category of data types is lifted to the following colimit of programs $\coprod_i (P_i \sqcup \langle \coprod_j R_j \rangle)$;
- the discrete lift of a data type is the program with the empty set of reactions.

7.5.8 EXERCISE

Workout the full characterisation of the category of Gamma programs and prove that it is indeed coordinated over the data types.