

4 UNIVERSAL CONSTRUCTIONS

In this chapter, we finally start getting into the real matter of Category Theory! The previous chapters just set the scene by providing basic definitions, methods, and examples. It is now time to show what can be obtained by using Category Theory instead of any other mathematical domain.

We have already hinted to the fact that Category Theory provides a totally different approach to the characterisation of a given domain of objects, namely to the fact that objects are characterised by their "social life", or "interactions", as captured by morphisms. This is precisely the point that we are going to exemplify in this chapter: how certain objects, or constructions, can be characterised in terms of standard relationships that they exhibit with respect to the rest of the universe (of relevant constructions). It is in this sense that these constructions are called "universal". At the same time, we will start shifting our focus from the social life of individual objects to that of groups or societies of interacting objects.

Indeed, this is where we start shifting some of the emphasis from the manipulation of objects to diagrams as models of complex systems. We have already mentioned (and will continue to do so) that one of our goals with this book is to show that Category Theory can go a long way in supporting the formalisation of software development methods and techniques that address the complexity of building and evolving systems. Central to this view is the notion of diagram as an expression of what is often called a configuration, be it the configuration of a running system as a network of simpler components, the way a complex program (text) or specification is put together from modules, the inheritance structure according to which program modules are organised, etc. This is also where Goguen's famous dogma [56] *"given a category of widgets, the operation of putting a system of widgets together to form a super-widget corresponds to taking a colimit of the diagram of widgets that shows how to interconnect them"* applies. We would like to take it as a motto for this chapter, if not the whole book!

This chapter is only an entry point to the ways Category Theory can address the complexity of system development. Subsequent chapters will introduce further constructions, techniques, and applications. Some readers may feel some frustration in that the examples that we use for illustrating these aspects – Eiffel, processes and specifications – are developed in a piecemeal way. Indeed, they are like threads that will have to be followed throughout the book to get the overall picture. The reason is that they are used in the book in support for the exposition and, hence, each piece is introduced only when and where it was judged to be the most useful. The only application that is developed as a whole is provided in Part Three – CommUnity and Software Architectures. Bear in mind that, at the end, the topic will be far from exhausted. The reader is encouraged to consult the references that are provided throughout.

4.1 Initial and terminal objects

At least at first-sight, the first universal constructions that we define are not so much “constructions” but “identifications” of distinguished objects. What distinguishes them from the other objects are what are usually called “universal properties”, a term that is associated, and used interchangeably in the literature, with “universal construction”.

4.1.1 DEFINITION – initial objects

An object x of a category \mathbf{C} is said to be *initial* iff, for every object y of \mathbf{C} , there is a unique morphism from x to y . A generic initial object is usually denoted by 0 . The unique morphism from an initial object 0 into an object x is denoted by 0_x .

4.1.2 PROPOSITION

1. Any two initial objects are isomorphic.
2. Any object isomorphic to an initial object is also initial.

PROOF

1. Let x and y be initial objects, and $f:x \rightarrow y$ and $g:y \rightarrow x$ the associated morphisms given by their universal properties. Consider now the morphism $(f;g):x \rightarrow x$. Because x is initial, we know that there is only one morphism with x as source and target. Hence, $(f;g)=id_x$. The same reasoning applied to y gives us $(g;f)=id_y$, showing that x and y are isomorphic.
2. Left as an exercise.

Hence, we usually refer to *the* initial object of a category, if one exists.

4.1.3 EXAMPLE

1. In **SET**, the initial object is the empty set. This is because the empty set can be mapped to any other set in a unique way: through the empty function. This also justifies the notation 0 usually adopted for initial objects.
2. In **LOGI**, the initial object is \perp (any contradiction). This is because anything can be derived from a contradiction. This again is consistent with the use of the notation 0 (as the truth value for false) for initial objects.
3. In **PAR**, the initial object is also the empty set: all functions from the empty set are, in fact, total, hence it is not surprising that we get the same initial object than in **SET**.
4. In **SET_D**, the initial objects are the singletons $\langle \{a\}, a \rangle$. This is because, according to the definition, there is one and only one way of mapping the designated ob-

ject of a pointed set: to the designated element of the target pointed set. Notice that, although \mathbf{SET}_0 , was built over \mathbf{SET} , the initial objects of the two categories do not coincide. Indeed, when adding structure to a given category to make another category, the universal constructions may change precisely because the structure has been changed. Nevertheless, the spirit of “emptiness” is still there – $\langle \{a\}, a \rangle$ is empty of “proper” elements.

4.1.4 DEFINITION – terminal objects

An object is terminal in a category \mathbf{C} iff it is initial in \mathbf{C}^{op} . That is, x is terminal in \mathbf{C} iff, for every object y of \mathbf{C} , there is a unique morphism from y to x . A generic terminal object is usually denote by 1 and the unique morphism from an object x is denoted by 1_x .

Once again, terminal objects are isomorphic and, therefore, we usually refer to *the* terminal object of a category, if one exists.

4.1.5 EXAMPLES

1. In \mathbf{SET} , the terminal objects are the singletons. This is because there is one, and only one way of mapping any given set to a singleton: by mapping all the elements of the source set (event if there is none...) to the element of the singleton. Any set with more than one element is clearly not terminal because it provides a choice for the target image of any element of the source set, i.e. it does not satisfy the uniqueness criterion. The empty set is not terminal because it only admits total functions from itself, i.e. it does not satisfy the existence criterion.

Notice that all singletons are indeed isomorphic in \mathbf{SET} . This is consistent with what we said before about the way Category Theory handles sets as objects: because we are not allowed to look into a set to see what elements it has, there is no way we can distinguish two singletons in terms of their structural properties. This also justifies the use of the notation 1 for arbitrary terminal objects.

On the other hand, any singleton can be used to identify the different elements of any non-empty set A by noticing that each element $a \in A$ defines, in a unique way, a function $a: 1 \rightarrow A$. Indeed, following the idea that morphisms characterise the “social life” of the objects of a category, the social relationships that a singleton set can establish with an arbitrary non-empty set characterise precisely the elements of that set.

This idea can be generalised to any terminal object $1_{\mathbf{C}}$ in an arbitrary category \mathbf{C} as a mechanism for identifying what, in Category Theory, are called points [74], constants [12], or global elements [22] of an object x : morphisms of the form $1 \rightarrow x$.

2. In \mathbf{LOGI} , the terminal object is \top (any tautology). This is because any tautology can be derived from any other formula. Again, this is also consistent with the use of the notation 1 (the truth value for true) for terminal objects.
3. In \mathbf{PAR} , the the terminal object is also the empty set. Indeed, there is always one and only one way of mapping any set to the empty one: through the partial function that is undefined in all the elements of the source!

Notice that, in spite of being a subcategory of **PAR**, **SET** has different terminal objects. On the one hand, the undefined function not being total, the empty set cannot play the role of terminal object in **SET**. On the other hand, because sets in **PAR** have "a richer social life", i.e. more morphisms, the singletons in **PAR** relate differently to other sets than they do in **SET**; in particular, besides the constant (total) function, they also admit the (partial) undefined one. Indeed, categories cannot be expected to share the same kind of initial/terminal objects with their subcategories, unless there is some special structural property that justifies so (see exercise below).

Hence, in **PAR**, the initial and the terminal objects coincide. Objects of an arbitrary category that are both initial and terminal are sometimes called null [79] or zero [1] objects.

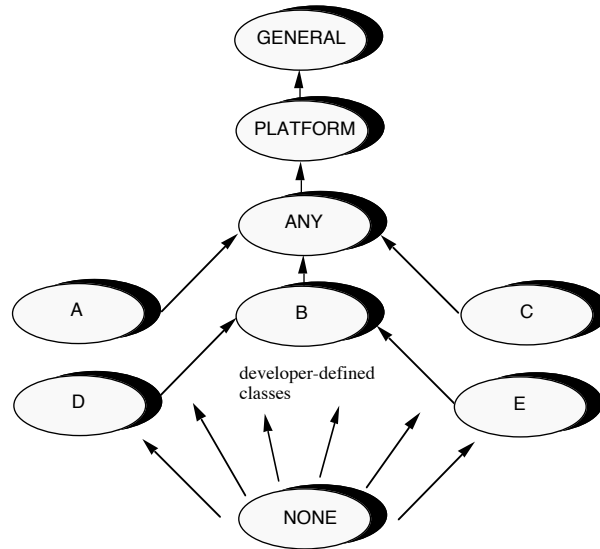
4. In \mathbf{SET}_D , the terminal objects are also the singletons $\langle \{a\}, a \rangle$. Hence, in \mathbf{SET}_D , the initial and the terminal objects also coincide.

4.1.6 EXERCISE

1. Let \mathbf{C} be an arbitrary category and $a \in \mathbf{C}$. Show that id_a is initial in a/\mathbf{C} . Further show that if a is terminal in \mathbf{C} , id_a is terminal in a/\mathbf{C} .
2. Show that the category \mathbf{SET}_D of pointed sets (3.2.1) "corresponds" to the comma category $1/\mathbf{SET}$ (3.2.2). This correspondence will be shown later on to define an isomorphism of categories. What can be said about the category $\mathbf{SET}/1$ (3.2.3)?
3. Show that, in any pre-order (2.2.5), initial objects coincide with the minimum and terminal objects coincide with the maximum, if they exist.
4. Show that, if \mathbf{D} is a full subcategory of \mathbf{C} , any initial (resp. terminal) object of \mathbf{C} that is an object of \mathbf{D} is also initial (resp. terminal) in \mathbf{D} .

4.1.7 EXAMPLE – Eiffel's inheritance structure

The inheritance structure of Eiffel has an initial object – the class NONE that inherits from every other class, and a terminal object – the class GENERAL from which every other class inherits.



4.2 Sums and products

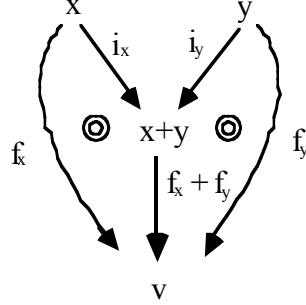
Intuitively, the universal constructions that are the theme of this chapter concern the possibility of finding objects that are able to capture the social lives of whole collections of objects and morphisms showing how the objects relate to one another. This is because, for instance, we are interested in the study of properties of whole systems, e.g. emergent behaviour, rather than isolated objects. The obvious questions that we have to answer is “what exactly is meant by a *collection*” and “what is the social life of such a collection of interacting objects”...

Leaving the first question out of the discussion for a while, and relying on an intuitive level of understanding for the time being, we start with a simple example. Not the simplest, though: this would be for collections consisting of only one object, in which case only isomorphic objects would have a social life that is able to capture the one of the given object... The simplest non-trivial example is that of a collection of two objects with no interactions between them, for instance two processes running in parallel with no communication between them, or two software modules with no dependencies between them.

Consider first the characterisation of the “out-going” communication for such a collection x, y of objects. We will say that, as a collection, x and y interact with other objects v via morphisms $f_x: x \rightarrow v$, $f_y: y \rightarrow v$. For instance, to say how a software module v uses x and y collectively, we have to say how it uses each of them in particular. This is because there are no dependencies between x and y , otherwise we would have to express the fact that these dependencies are respected.

An object that is able to stand for the relationships that a collection x, y of objects has towards its environment is called their sum and denoted by $x+y$. In fact, the sum is more than an object. We need to make explicit how x and y relate to it. This requires morphisms (injections) $i_x: x \rightarrow x+y$ and $i_y: y \rightarrow x+y$. The ability for this object and the connecting morphisms to characterise the relationships that the collection has towards its environment can then be expressed by the property that any such inter-

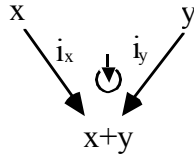
action $f_x: x \rightarrow v$, $f_y: y \rightarrow v$ can be performed via $x+y$ in the sense that there is a unique morphism $k: x+y \rightarrow v$ through which f_x and f_y can be intercepted, i.e. $i_x; k = f_x$ and $i_y; k = f_y$. The morphism k is often represented by $f_x + f_y$.



4.2.1 DEFINITION – sum

Let \mathbf{C} be a category and x, y objects of \mathbf{C} . An object z is said to be a *sum* (or *coproduct*) of x and y with injections $i_x: x \rightarrow z$ and $i_y: y \rightarrow z$ iff for any object v and pair of morphisms $f_x: x \rightarrow v$, $f_y: y \rightarrow v$ of \mathbf{C} there is a unique morphism $k: z \rightarrow v$ in \mathbf{C} such that $i_x; k = f_x$ and $i_y; k = f_y$.

In order to express, in diagrams, that the objects and morphisms shown are related by a universal construction, we use the symbols \circlearrowright and \circlearrowleft according to rules that we will make explicit in each case. For instance, in the case of sums,



As it should be expected, the exact identity of the sum should be of no importance, just the way in which it relates to the other objects. Hence, the following property holds:

4.2.2 PROPOSITION

If a sum of x and y exists, it is unique up to isomorphism and is denoted by $x+y$.

PROOF

Let z be a sum of x and y with injections $i_x: x \rightarrow z$ and $i_y: y \rightarrow z$ and w another sum of x and y with injections $j_x: x \rightarrow w$ and $j_y: y \rightarrow w$. By the universal property of z , we can conclude that there is a morphism $k: z \rightarrow w$ such that $i_x; k = j_x$ and $i_y; k = j_y$. Applying the same reasoning to w , we conclude that there is a morphism $l: w \rightarrow z$ such that $j_x; l = i_x$ and $j_y; l = i_y$. Consider now the morphism $(k; l): z \rightarrow z$. It satisfies:

$$\begin{aligned}
 i_x; (k; l) &= (i_x; k); l && \text{associativity of ;} \\
 &= j_x; l && \text{universal property of } i_x \\
 &= i_x && \text{universal property of } j_x
 \end{aligned}$$

Similarly, we can prove that $i_y;(k;l)=i_y$. The universal property of z , i_x and i_y implies that there is only one morphism $z \square z$ satisfying this property. Hence, $(k;l)=id_z$. Using the universal property of w , j_x and j_y we can prove in exactly the same way that $(l;k)=id_w$. Therefore k is an isomorphism.

4.2.3 EXAMPLE – logical disjunction

1. In **LOGI**, sums correspond to disjunctions.
2. In **PROOF**, sums also correspond to disjunctions. Indeed, the sum of A and B is a sentence characterised by morphisms that capture the introduction and elimination rules for the disjunction as a logical operator:

$$\begin{array}{ccc} i_A:A \square A & B & i_B:B \square A & B \\ f_A:A \square C, \square f_B:B \square C & & \\ \hline f_{A+B}:A \square B \square C & & \end{array}$$

Notice that the conditions required of these three morphisms (derivations) correspond to properties of normalisation that are typical of proof-theory.

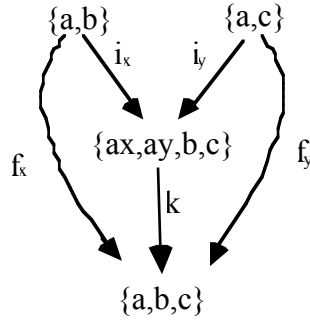
4.2.4 EXAMPLE – disjoint union of sets

In the category **SET**, the disjoint union $x \oplus y$ (with corresponding injections) is the sum of x and y .

PROOF

- existence: consider an arbitrary object v and pair of morphisms $f_x:x \square v$, $f_y:y \square v$. Define $k:x \oplus y \square v$ as follows: given $A \square x \oplus y$, let $k(A)=f_x(a)$ if $A=i_x(a)$ with $a \square x$ and $k(A)=f_y(a)$ if $A=i_y(a)$ with $a \square y$. This is a proper definition of a total function because, on the one hand, every element of $x \oplus y$ is either in the image of x through i_x or the image of y through i_y and, on the other hand, these two images are disjoint (which removes any conflict of choice between which case to apply). The conditions $i_x;k=f_x$ and $i_y;k=f_y$ are satisfied by construction.
- uniqueness: given any other total function $k':x \oplus y \square v$, the conditions $i_x;k'=f_x$ and $i_y;k'=f_y$ define k' completely (and equal to k).

In order to illustrate the construction and show why the union of sets is not (always) their sum, consider the following example where f_x and f_y are set inclusions. The injections are such that $i_x(a)=ax$ and $i_y(a)=ay$. By construction, k is such that $k(ax)=k(ay)=a$. The union $\{a,b,c\}$ does not provide a sum through the corresponding inclusions because there is no total function $l:\{a,b,c\} \square \{ax,ay,b,c\}$ that satisfies the commutativity conditions. Indeed, these require $l(a)=ax$ because $ax=i_x(a)$ and $l(a)=ay$ because $ay=i_y(a)$.



Intuitively, this happens because the union is an operation that “looks” inside the sets to which it is being applied in order not to repeat the elements that they have in common. That is to say, it is an operation that relies on an interaction between the sets to which it applies, whereas the sum is defined over objects without any relationship between them.

As we have already mentioned, but still worth repeating because it is one of the aspects of Category Theory that, in our opinion, has a strong “engineering” relevance, especially in the context of service-oriented development, all interactions need to be made explicit and external to the entities involved. That is to say, we cannot rely on implicit relationships such as the use of the same names in the definition of different objects. This may sound too “bureaucratic” but we are far from suggesting that Category Theory should be used directly, “naked”, as a language for specification, modelling or, even, programming. We view Category Theory as a mathematical framework that provides support for those activities. Hence, the need for such explicit and external representation of all interactions is, in our opinion, a bonus because it enforces directly fundamental properties of the methodology that is being supported, service-oriented in the most general case, but also component or object-oriented as amply demonstrated in the literature.

In the next section, we address constructions that involve interactions. In the rest of this section, we look at the dual construction of sums: products. We do so extensionally, i.e. by providing the definition of the construction directly. The reason for doing so in spite of knowing already that it is enough to reverse the direction of the arrows, is that, from a “pragmatic” point of view, one is “naturally” biased by the direction of the arrows and, in certain contexts, tradition is that one uses products instead of sums. Our experience in using and teaching Category Theory is that big arguments and misunderstandings are too often caused by being presented with the duals of constructions that otherwise would be very familiar! Hence, it is important that we know these more basic constructions explicitly in both forms rather than having to translate back and forth every time between a category and its dual.

4.2.5 DEFINITION – product

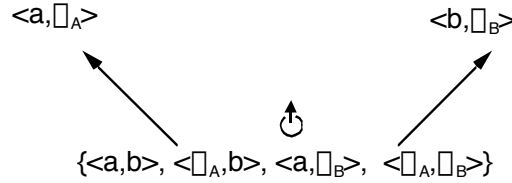
The dual notion of sum is *product*. That is to say, letting \mathbf{C} be a category and x, y objects of \mathbf{C} , an object z is said to be a product of x and y with projections $\pi_x: z \rightarrow x$ and $\pi_y: z \rightarrow y$ iff for any object v and pair of morphisms $f_x: v \rightarrow x$, $f_y: v \rightarrow y$ of \mathbf{C} there is a unique morphism $k: v \rightarrow z$ in \mathbf{C} such that $k; \pi_x = f_x$ and $k; \pi_y = f_y$.

Products handle the relationships from the environment towards collections of two unrelated objects. It is easy to see that, in **SET**, the product of two sets is given, up

to isomorphism, by their Cartesian product and, in **LOGI**, products capture conjunction.

4.2.6 EXAMPLE – parallel composition without interactions

In the category **SET**_↓ of pointed sets, products are constructed in the same way as in **SET**. The Cartesian product of two pointed sets includes all pairs of “proper elements”, the pairs of which one and only one of the elements is a designated one, and the pair of designated elements. We claim that, together with the projections, this Cartesian product is still a product of the pointed sets. On the one hand, it is trivial to prove that if we elect the pair of designated elements as the designated element of the product, we obtain through the original **SET**-projections, morphisms of pointed sets. On the other hand, the commutativity requirements make sure that all universal functions $f_x \sqcup f_y$ are morphisms of pointed sets, i.e. preserve designated elements. We will return to this “style” of proof argument of universal properties for certain categories with “structure” later on in the book.



If we consider the use of pointed sets for modelling the alphabets of concurrent processes, as suggested in 3.2.1, i.e. each “proper” element represents an event whose occurrence may be witnessed during the lifetime of the process and the designated element represents an event of the environment, products give us all the possible events in which both processes participate, plus all possible events in which only one process participates, together with the “silent” environment event in which none of the processes participates.

In the specific case in which we identify events with synchronisation sets of method execution, the product provides for joint executions as unions of synchronisation sets. To be more precise, when we work in **POWER** (see 3.3.2), we can take as a representation of a pair $\langle a, b \rangle$ the set $a \sqcup b$. This is because, as sets, the Cartesian product $2^A \sqcup 2^B$ is isomorphic to $2^{A \oplus B}$ and the isomorphism is given, precisely, by the map that associates pairs $\langle a, b \rangle$ with the set $a \oplus b$. This is to show that, because universal constructions are only unique up to isomorphism, we can choose from the isomorphism class the representation that suits us best. In this case, it justifies a uniform treatment of concurrent process alphabets as synchronisation sets.

In summary, through products, we obtain the alphabet of the process that is the interleaving of the given ones.

4.2.7 EXERCISE

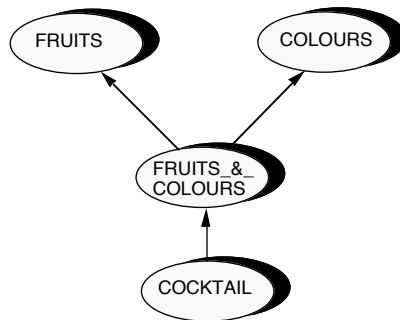
Characterise products and sums of partial functions (i.e. in **PAR**). Show in particular that sums work like in **SET**, i.e. they compute the disjoint union, but that products, besides the pairs that result from the cartesian product, include the disjoint union of the two sets as well. That is to say, products in **PAR** are very “similar” to those in

SET_I. Why do you think this is so? What about sums in **SET**_I? How do they compare?

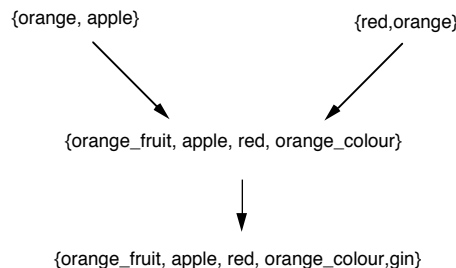
This example is also another good illustration of the fact that universal constructions do not carry necessarily from a category to its subcategories.

4.2.8 EXAMPLE – inheritance without name clashes

In Eiffel, products capture "minimal" inheritance without name clashes, a typical example being the following:



We know already that, when the sets of features of the classes in an inheritance graph are considered, the arrows are reversed, so the universal construction on the underlying features is a sum:



The “automatic renaming” associated with this categorical construction is very useful because it makes sure that no confusion arises from the inadvertent use of the same names for different “things” in different contexts. This is why the injections/projections are very much part of the concept of sum/product: they keep a record of the renamings that take place, i.e. of “who is who” or, better, “who comes from where”. However, in many situations, we want to make “joins”, i.e. identify things that are meant to be the same but were included in different contexts. This is the purpose of the universal construction that we illustrate next.

4.3 Pushouts and pullbacks

We already know that, in Category Theory, by default, things are different even if they were given the same names in different contexts. How, then, can we say that

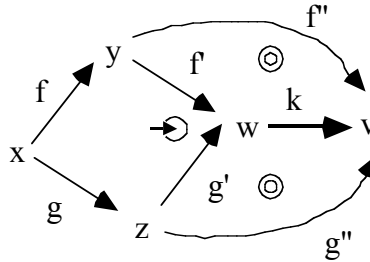
some things are the same? For instance, how to require that two processes synchronise on given events? How to require that features declared in different object classes be identified when performing multiple inheritance?

A distinguishing factor of Category Theory, and one that makes it so suitable for addressing architectural concerns and service-oriented development in Software Engineering, is that such forms of interaction are *exogenous*, i.e. they have to be established *outside* the objects involved. For instance, interactions in object-oriented development are *indogenous* because feature calling (clientship) is embedded explicitly in the code of the caller (client). In Category Theory, the means that we have for establishing interactions is through third objects that handle communication between the ones that are being interconnected via given morphisms. In the case of sums, we are dealing with interactions at the level of the sources, i.e. we are interested in the social life of pairs of morphisms $f:x \rightrightarrows y$, $g:x \rightrightarrows z$, and in the case of products, the interactions are on the target side – $f:y \rightrightarrows x$, $g:z \rightrightarrows x$.

4.3.1 DEFINITION – pushout

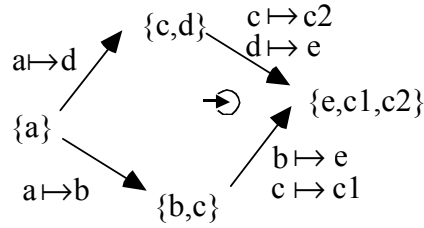
Let \mathbf{C} be a category and $f:x \rightrightarrows y$, $g:x \rightrightarrows z$ morphisms of \mathbf{C} . A pushout (or amalgamated sum) of f and g consists of two morphisms $f':y \rightrightarrows w$ and $g':z \rightrightarrows w$ such that

- $f;f'=g;g'$
- for any other two morphisms $f'':y \rightrightarrows v$ and $g'':z \rightrightarrows v$ such that $f;f''=g;g''$, there is a unique morphism $k:w \rightrightarrows v$ in \mathbf{C} such that $f';k=f''$ and $g';k=g''$.



4.3.2 EXAMPLE – amalgamated sums of sets

In **SET**, pushouts perform what are usually called “amalgamated sums”, i.e. they allow us to identify (join) elements as indicated by the “middle object” and corresponding morphisms. For instance, in the example below, the morphisms indicate that elements b and d are to be identified. Because nothing is said about c , the categorical default applies: the two occurrences are to be distinguished because the fact that the same name was used is accidental.



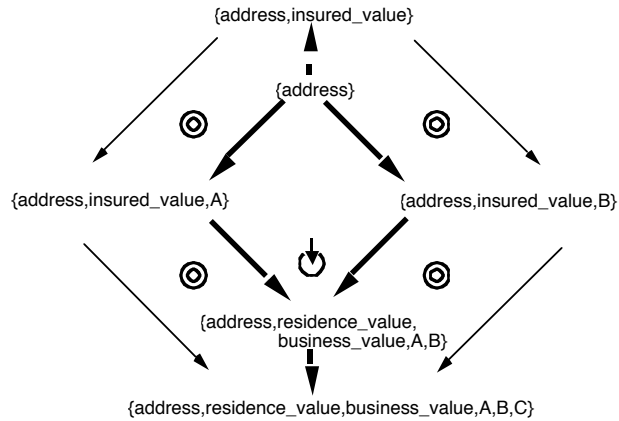
The proof that what we have built is actually a pushout can be outlined as follows. On the one hand, the commutativity requirement is clearly satisfied. On the other hand, for the universal property, consider given functions $f: \{c,d\} \rightarrow A$ and $g: \{b,c\} \rightarrow A$ satisfying the commutativity requirement: $f(d)=g(b)$.

Any function $k: \{e,c1,c2\} \rightarrow A$ that satisfies the commutativity requirements of the universal property must be such that $k(e)=f(d)$, $k(e)=g(b)$, $k(c1)=g(c)$ and $k(c2)=f(c)$. These requirements leave no other choice for defining k , hence uniqueness is ensured. Because these requirements are consistent, which is due to the fact that $f(d)=g(b)$, existence is also ensured.

More interesting than this proof is the mechanism through which pushouts can be systematically constructed in **SET**. This will be revealed once we address the whole process of computing pushouts in more general terms.

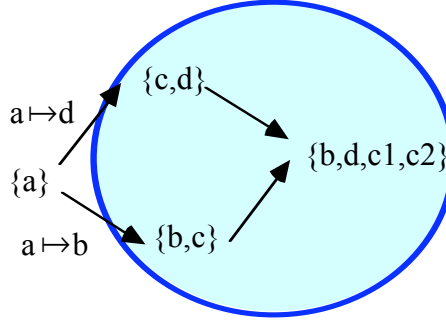
4.3.3 EXAMPLE – multiple inheritance in Eiffel

Pushouts are the universal construction that allows us to join (merge) features during multiple inheritance. For instance, the construction of HOME_BUSINESS requires that the home and business addresses be merged:

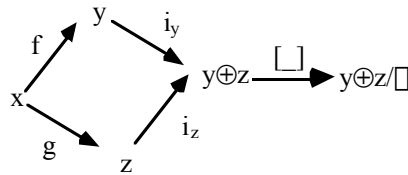


Let us now return to the definition of pushout and explain how the interactions operate. The definition consists of two main requirements: a commutativity condition and a “universal” property. The universal property is just an instance of a “ritual” that we shall explain in the next section. For the time being, notice how similar it is to the corresponding property of sums, and try to see where it lies in the definition of initial object...

The commutativity requirement is the one responsible for the “amalgamation” or, more generally, for encapsulating the interactions in place. In the case of **SET**, the amalgamation takes place as a quotient defined over the sum of the objects for an equivalence relation that is defined by the middle object and the connecting morphisms. If we take two morphisms $f:x \rightarrow y$, $g:x \rightarrow z$ and we can find a sum for y and z , we obtain a square like before except for the fact that it does not necessarily commute. For instance,



Commutativity fails because, if we start by picking an arbitrary element in the middle object and follow all the paths from it, we do not arrive at the same element. For instance, in the specific example that we have just used, there are two such paths, one of which terminates in $c1$ and the other in $c2$. In order to enforce the commutativity requirement, we have to “equalise” the ends of these paths. Category Theory does not perform miracles (yet), so it cannot make equal entities that are actually different...What we usually do in Mathematics is to define an equivalence relation expressing that, albeit being different, these endpoints should be considered the same. In **SET**, this equivalence relation is formally defined as being generated from all pairs $f(i_y(a)) \sim g(i_z(a))$ where $a \in x$. We claim that the quotient set of the sum $y \oplus z$ by this equivalence relation together with the functions $[i_y(_)]$ and $[i_z(_)]$ that, to the elements of the given sets, assigns them their equivalence classes, is a pushout of f and g .



The proof of this result is not that difficult because the whole construction was made to ensure commutativity, i.e. $[i_y(f(a))] = [i_z(g(a))]$ for every $a \in x$ which is precisely the set of pairs $f(i_y(a)) \sim g(i_z(a))$ that generate the equivalence relation. The universal property can now be derived from the universal properties of the sum and quotients: consider any other two morphisms $f':y \rightarrow v$ and $g':z \rightarrow v$ such that $f'f'' = g'g''$. We know that there is a unique morphism $k:y \oplus z \rightarrow v$ such that $i_y;k = f''$ and $i_z;k = g''$. From the properties of quotients we know that there is a unique way in which this map can be factorised through $[]$ i.e. there is a unique $k':y \oplus z / \sim \rightarrow v$ such that $k = [] ; k'$. Using associativity of morphism composition, this provides us with the required unique morphism satisfying $(i_y;[]);k' = f''$ and $(i_z;[]);k' = g''$.

This is how amalgamated sums work on sets. How can we generalise the quotient to categories in general? Taking the diagram above to be over an arbitrary category \mathbf{C} , the purpose of the quotient is to make the following diagram commute:

$$\begin{array}{ccc} & f, i_y & \\ x & \xrightarrow{\quad} & y \oplus z \\ & g, i_z & \end{array}$$

The idea is to do so via a third morphism $e: y \oplus z \rightarrow v$ whose purpose is to replace the initial equality $f; i_y = g; i_z$ by $(f; i_y); e = (g; i_z); e$. However, there are many ways of doing so. One of them is to choose v to be a terminal object (if one exists), but this is clearly too intrusive on the given morphisms because it over-equalises them. We would prefer to do it in a minimal way, which is what the universal property of quotients provides: for any other morphism $e': y \oplus z \rightarrow w$ that also equalises the morphisms, i.e. such that $(f; i_y); e' = (g; i_z); e'$, there should be a unique $k: v \rightarrow w$ such that $e; k = e'$.

$$\begin{array}{ccccc} & f, i_y & & & \\ x & \xrightarrow{\quad} & y \oplus z & \xrightarrow{\quad e \quad} & v \\ & g, i_z & & \searrow e' & \downarrow k \\ & & & & w \end{array}$$

This is precisely another instance of a universal construction:

4.3.4 DEFINITION – co-equaliser

Let \mathbf{C} be a category and $f: x \rightarrow y, g: x \rightarrow y$ morphisms of \mathbf{C} . A *co-equaliser* of f and g consists of a morphism $e: y \rightarrow z$ such that

- $f; e = g; e$
- for any other morphisms $e': y \rightarrow v$ such that $f; e' = g; e'$, there is a unique morphism $k: z \rightarrow v$ in \mathbf{C} such that $e; k = e'$.

Hence, pushouts can be obtained from sums and co-equalisers.

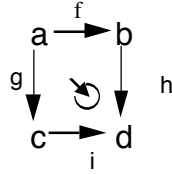
4.3.5 EXERCISE

1. Formalise the observation above about pushouts being obtained from sums and co-equalisers and prove it.
2. Prove that co-equalisers are a particular case of pushouts.
3. Prove that, if initial objects exist, sums can be obtained from pushouts.

There are a number of properties that are both typical and useful. They are left here as exercises, with a strong recommendation.

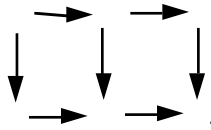
4.3.6 EXERCISE

1. Prove that the universal arrow e in a co-equaliser is epic (thus generalising the fact that a quotient map is surjective).
2. Consider a pushout diagram



Prove that, if g is epic, so is h .

3. Consider a commutative diagram



Prove that if both internal squares are pushouts so is the external rectangle, and that if the external rectangle and the left square are pushouts, so is the right square.

Let us consider now the dual of pushouts.

4.3.7 DEFINITION – pullback

Let \mathbf{C} be a category and $f: y \rightarrow x$, $g: z \rightarrow x$ morphisms of \mathbf{C} . A *pullback* (or fibred product) of f and g consists of two morphisms $f': w \rightarrow y$ and $g': w \rightarrow z$ such that

- $f'g' = f;g$
- for any other two morphisms $f'': v \rightarrow y$ and $g'': v \rightarrow z$ such that $f''g'' = f;g$, there is a unique morphism $k: v \rightarrow w$ in \mathbf{C} such that $k;f' = f''$ and $k;g' = g''$.

Pullbacks can also be explained from products and a construction that equalises arrows, except that, this time, the equalising needs to be made on the source and not the target side of the arrows. Not surprisingly, this universal construction is named equaliser, the dual of the co-equalisers.

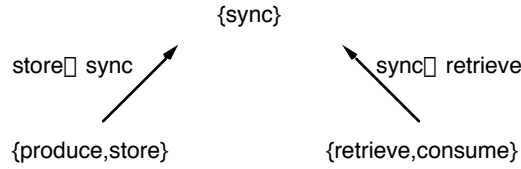
Whereas, in set-theoretic terms, we saw that equalising on the target side corresponds to taking a quotient to group entities that should be the same in the same equivalence relation, equalising on the source side is even conceptually easier: it is enough to restrict the domain by throwing away the elements over which the two functions disagree. That is, we equalise through the inclusion $\{a \in y: f(a) = g(a)\} \rightarrow y$.

$$\{a \sqcup y: f(a)=g(a)\} \xrightarrow{m} y \begin{array}{c} \xrightarrow{f} x \\ \xrightarrow{g} x \end{array}$$

Hence, computing a fibred product consists in computing a product followed by a “purge” of the pairs that violate the commutativity requirement. We are going to illustrate this procedure with process alphabets.

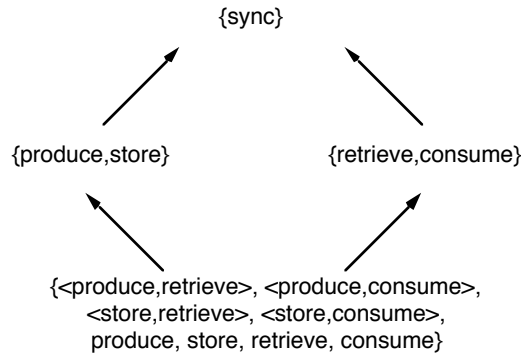
4.3.8 EXAMPLE – parallel composition of processes with interactions

We have already argued that in the category \mathbf{SET}_I of pointed sets, products are constructed in the same way as in \mathbf{SET} through Cartesian products. When the pointed sets capture process alphabets, we saw that this construction captures parallel composition without synchronisation in the sense that all the pairs of events are generated in an interleaving semantics. Fibred products allow us to compute parallel composition with synchronisation constraints. For instance, consider two process alphabets $\langle \{produce, store\}, \square_P, \square_P \rangle$ and $\langle \{consume, retrieve\}, \square_C, \square_C \rangle$. The alphabet of the parallel composition of the two processes when required to synchronise in the *store* and *retrieve* events can be obtained through the pullback of:



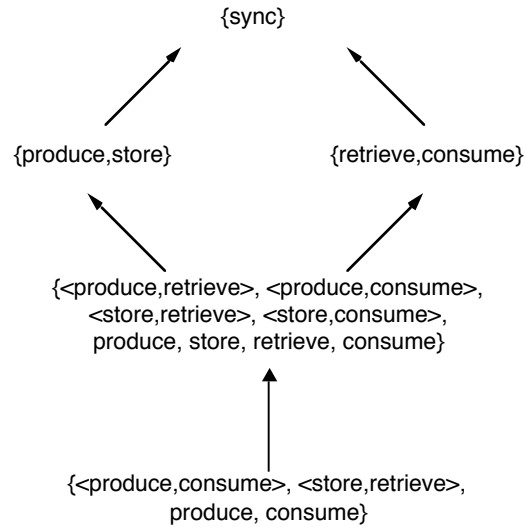
For simplicity, we have omitted the designated elements from the representations of the pointed sets and will only show the proper events. The middle object models the alphabet of the communication channel that is being used to interconnect the two processes. The maps mean that *store* and *retrieve* are required to be synchronised.

The product of the two alphabets gives us:



Recall that the original events are obtained as a result of synchronisations with the designated event, thus capturing system events in which only one process participates.

Notice that the diagram does not commute. For instance, the system event $\langle store, consume \rangle$ is mapped, on the left, to *sync* but, on the right, to the designated event of the channel. In order to make the diagram commute, we throw away all system events on which the maps to the channel do not agree:



Basically, the events that remain are all possible combinations of executing *produce* and *consume* because there is no synchronisation constraint on them, plus the synchronisation that is explicitly required.

4.3.9 EXERCISE

Follow-up on 4.2.7 by characterising fibred products and amalgamated sums of partial functions (i.e. in **PAR**) and relating them to **SET** and **SET_∇**.

4.3.10 EXERCISE

1. Define explicitly the notion of equaliser and prove that pullbacks can be obtained from products and equalisers as suggested.
2. Prove that equalisers are a particular case of pullbacks.
3. Prove that, if terminal objects exist, products can be obtained from pullbacks.
4. Prove that the equalising arrow m is a mono.

4.4 Limits and colimits

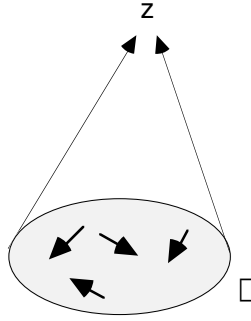
It should be clear by now that the notion of “collective behaviour” that we wish to capture through universal constructions takes diagrams as the expression of the collection of objects and interactions that constitute what we could call a “system”. In fact, we tend to use diagrams for dealing with “complex” entities for which the objects of the category provides components and the morphisms the means for inter-connecting them. Hence, for instance, a typical use of diagrams is for defining con-

figurations. The universal constructions that we are addressing in this chapter allow us to define the semantics of such complex entities by internalising the configuration and collapsing the structure into an object that captures the collective behaviour.

An aspect of these universal constructions that is important to keep in mind is the fact that they deliver more than an object: this object comes together with morphisms that relate it to the objects out of which it was constructed. It is through these morphisms that we can understand how properties of the system (complex object) emerge from the properties of its components and the interconnections between them. Hence, the constructions are better understood in terms of structures that consist of objects together with configurations to which they relate. These are called (co)cones.

4.4.1 DEFINITION – co-cone


Let $\square: I \rightarrow \mathbf{C}$ be a diagram in a category \mathbf{C} . A co-cone with base \square is an object z of \mathbf{C} together with a family $\{p_a: \square_a \rightarrow z\}_{a \in I_0}$ of morphisms of \mathbf{C} , usually denoted by $p: \square \rightarrow z$. The object z is said to be the vertex of the co-cone, and, for each $a \in I_0$, the morphism p_a is said to be the edge of the co-cone at point a . A co-cone p with base $\square: I \rightarrow \mathbf{C}$ and vertex z is said to be commutative iff for every arrow $s: a \rightarrow b$ of graph I , $\square_s p_b = p_a$.

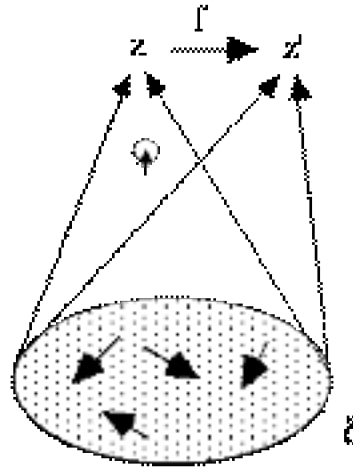


We tend to consider the family p of morphisms as identifying how the source is represented in, or a component of, the target – the object z . The commutativity property is important because it ensures that the interconnections that are expressed in the base through the morphisms are also represented in z . Hence, z is an object that is able to “represent” the base objects and their interactions. However, it may not do so in a “minimal” way. If one such minimal representation exists, we call it a colimit of the diagram.

4.4.2 DEFINITION – colimit

Let $\square: I \rightarrow \mathbf{C}$ be a diagram in a category \mathbf{C} . A colimit of \square is a commutative co-cone $p: \square \rightarrow z$ such that, for every other commutative co-cone $p': \square \rightarrow z'$, there is a unique morphism $f: z \rightarrow z'$ such that $p'_a = f \circ p_a$, i.e. $p'_a f = p_a$ for every edge.

Colimit co-cones are decorated with .



Notice how all the ingredients that were used in the universal constructions that we studied in the previous sections are present in this definition. On the one hand, a commutativity requirement. On the other hand, a universal property that ensures minimality.

4.4.3 EXERCISE

Show that initial objects, sums, co-equalisers and pushouts are instances of colimits by identifying the shape of the base diagrams and checking that the properties required are equivalent.

Co-cones over a given base can be organised in a category in an “obvious way”:

4.4.4 PROPOSITION

Let $\square: I \rightarrow \mathbf{C}$ be a diagram in a category \mathbf{C} . A category $\mathbf{CO_CONE}(\square)$ is defined whose objects are the commutative co-cones with base \square and the morphisms f between co-cones $p: \square \rightarrow z$ and $q: \square \rightarrow w$ are the morphisms $f: z \rightarrow w$ such that $p \circ f = q$, i.e. $p \circ f = q$ for every edge.

4.4.5 EXERCISE

1. Prove the previous result.
2. Prove that the colimits of a diagram \square are the initial objects of $\mathbf{CO_CONE}(\square)$.
3. Conclude that colimits are unique up to isomorphism.

4.4.6 DEFINITION – co-completeness

A category is (finitely) co-complete if all (finite) diagrams have colimits.

There are several results on the finite co-completeness of categories. A commonly used one is:

4.4.7 PROPOSITION

A category **C** is finitely co-complete iff it has initial objects and pushouts of all pairs of morphisms with common source.

Using this result and the examples that we studied in the previous sections, we can conclude, for instance, that both **SET** and **LOGI** are finitely co-complete.

These notions are also straightforward generalisations of well-known constructions over ordered sets, namely least upper bounds and greatest lower bounds.

A more “interesting” example can be given over Eiffel class specifications:

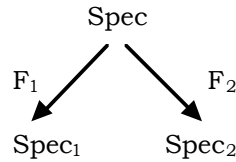
4.4.8 EXAMPLE – Eiffel's "join semantics" rule

The category **CLASS_SPEC** is finitely co-complete.

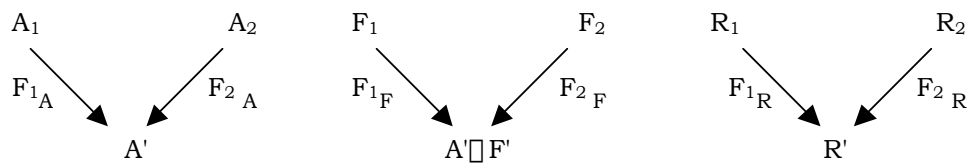
PROOF

Using the previous result, we show that **CLASS_SPEC** admits initial objects and pushouts.

1. It is easy to prove that the class specification that has no features is initial.
2. On the other hand, pushouts work as follows:
 - given a diagram



we first compute the pushout for the underlying diagram of signatures (sets of attributes, functions, and routines). This pushout returns



Recall that, according to the definition of morphism given in section 3.4, functions can be mapped to attributes (but not the other way around)! Hence, equivalence classes can mix together attributes and functions. The pushout classifies an equivalence class as an attribute iff it contains at least one attribute, and as a function iff it only contains functions. Routines are not mixed together with functions or attributes.

- for every routine $r' \sqcup R'$, we compute its pre/post-conditions as follows

$$\begin{aligned}
 &\text{if } r' = F_{1R}(r_1) = F_{2R}(r_2) \text{ then} \\
 &\quad pre_{r'} = \sqcap F_1(pre_{r_1}) \sqcap F_2(pre_{r_2}) \text{ and } post_{r'} = \sqcap F_1(post_{r_1}) \sqcap F_2(post_{r_2})
 \end{aligned}$$

if $r' = F_{1R}(r_1) \sqcup F_{2R}(r_2)$, then $pre_{r'} = \sqcup F_1(pre_{r_1})$ and $post_{r'} = \sqcup F_1(post_{r_1})$

if $r' = F_{2R}(r_2) \sqcup F_{1R}(r_1)$, then $pre_{r'} = \sqcup F_2(pre_{r_2})$ and $post_{r'} = \sqcup F_2(post_{r_2})$

- the new invariant is $I' = F_1(I_1) \sqcup F_2(I_2)$.

The proof that the maps F_1 and F_2 that result from these constructions are indeed morphisms of class specifications is trivial. The commutativity property is inherited from the pushouts in **SET** that determine the new attributes and routines. The universal property is left as an exercise and, basically, reflects the universal properties of conjunction and disjunction that we already identified in the previous sections.

Notice how this construction conforms with the **Join Semantics rule**, and inheritance of invariants via concatenation of parent invariants [88].

4.4.9 EXERCISE

Workout in detail the way pushouts operate on attributes and functions of class specifications.

4.4.10 DEFINITION

The dual notion of co-cone is **cone** and the dual notion of colimit is **limit**. Limit cones are decorated with \lim_{\leftarrow} .

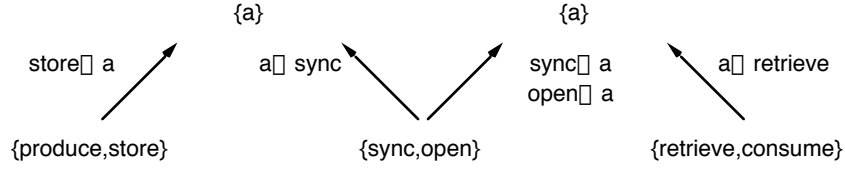
Hence, limits generalise terminal objects, products, equalisers and pullbacks. From the examples studied in the previous sections, we can also conclude that both **SET** and **SET_g** are finitely co-complete.

4.4.11 EXAMPLE – parallel composition of processes with interactions

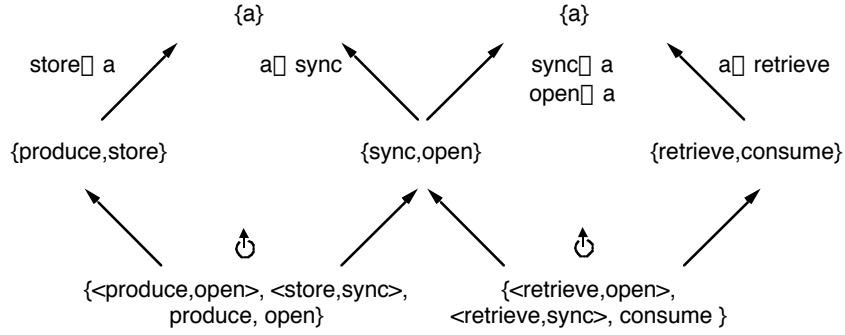
In order to illustrate the calculation of limits, consider process alphabets once again. In the previous section, we showed how we can synchronise a consumer and a producer on the *store/retrieve* events. This interconnection is, however, too tight because it ties the consumer completely to that producer and does not allow it to consume from other producers. Hence, the situation that we would like now to model is the one in which the consumer retrieves from a producer but leaves open the possibility of retrieving from other producers as well. This form of interconnection cannot be achieved simply through a channel as before.

What we have to do is to make explicit a communication protocol that can be placed in between the producer and the consumer. The alphabet of this protocol needs to account for the synchronisation between the consumer and the producer, which we model through an event *sync*, and the open communication between the consumer and other possible producers, which we model through an event *open*. Omitting, as before, the designated events, this configuration is given by the following diagram:

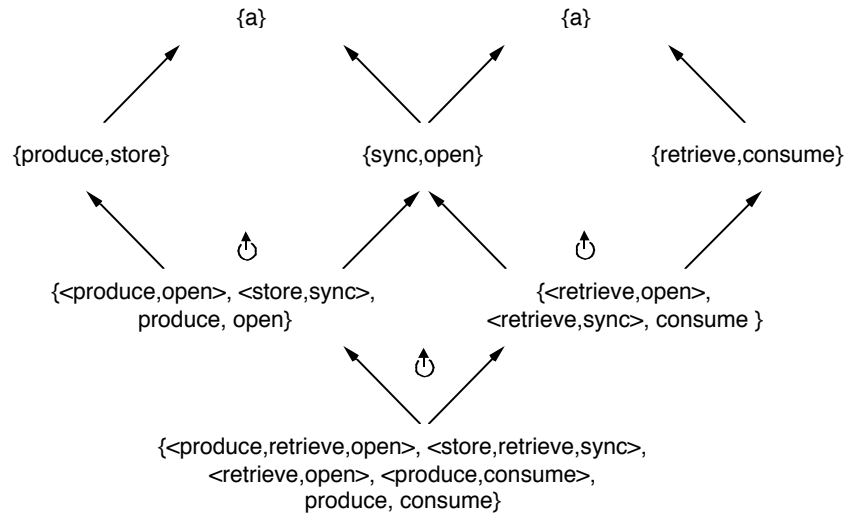
4. UNIVERSAL CONSTRUCTIONS



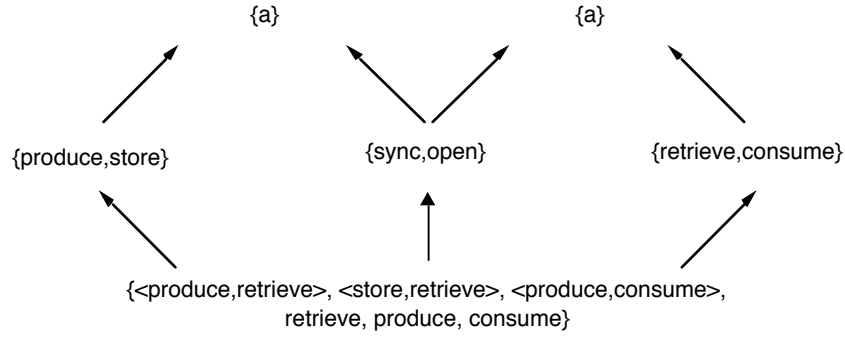
The dual of 4.4.7 can be used to compute the limit of this diagram by computing successive pullbacks. We start with the two obvious ones:



We finalise with the new obvious, central one:



Using the privilege of choosing among the isomorphic representations of the limit, we can simplify the notation by omitting the reference to the communication protocol (although at the expense of making the morphism less obvious: they are no longer projections). Hiding the intermediate constructions at the same time, we obtain the following commutative cone:



This representation makes it obvious that, with respect to the previous interconnection, we are now allowing the *retrieve* to occur independently of *store*, which justifies why it now appears as a single event (though synchronised with *open*) and synchronised with *produce* (and *open*).

Although we have systematically motivated universal constructions such as these in terms of parallel composition of processes, with and without interaction, we have remained at the level of the composition of process alphabets, i.e. we have not taken into account their behaviours. There are two reasons for that. On the one hand, as we shall see, all the complexity of interaction resides on alphabets. On the other hand, the way behaviours can be brought into the picture can be used to illustrate another categorical structure and construction. Hence, we defer the "completion" of this topic to later on (6.3.2 and 6.3.7).

For similar reasons, we defer the illustration of the application of universal constructions to configurations of specifications to later on (6.1.24) so that we can use it to illustrate another class of categorical structures and constructions.

