

SQL versus coSQL — a compendium to Erik Meijer’s paper


Maarten Fokkinga

version of October 3, 2012, 12:08

Introduction. In a seminal paper, Erik Meijer [6] enthusiastically shows an interesting relation between the well-known SQL and OO representations of facts from the real world. Phrased in terms of category theory, these turn out to be dualizations of each other (hence he speaks of SQL and coSQL), and many of their properties are in some sense dual to each other. This note enumerates most of his ideas in a quite different and more concise style, and might be used as a compendium to his paper.

I will leave out all blah blah that I consider not relevant; in particular, I don’t present syntactic expressions, for which Meijer uses C# and LINQ, but instead focus on semantics (drawn as pictures!). Also, I skip the following topic:

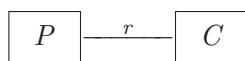
The potential benefit for the database industry, including the claim that the theoretical considerations of the paper might enable an economic growth.

Unfortunately, ACM’s type setting of Meijer’s paper has corrupted the lay-out of the syntactic fragments and drawings considerably: symbols like \Rightarrow and \in and pictorial boxes like  have sometimes been left out wrongly and the lay-out of program fragments has sometimes been corrupted.

This note first presents a running example, and then explains most of Meijer’s claims, observations, and thoughts in a series of numbered paragraphs that use the running example.

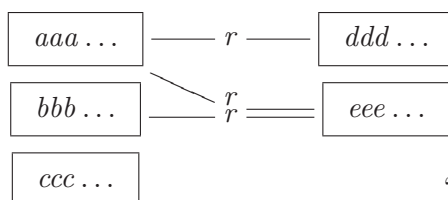
Running example

The general case. Consider the following typical fragment of an ERD:



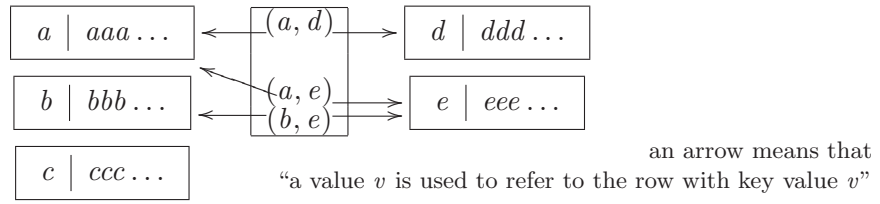
(One might interpret this fragment as modelling *Professors* being *involved* in *Courses*.)

Consider an instantiation having three *P* and two *C* instances, with the following relationship:

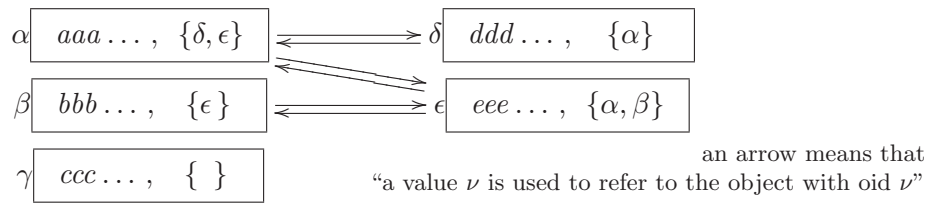


$x -r- y$ means
“(x, y) is a member of relation r ”

In the SQL representation the programmer has to make sure that each instance has a key attribute, say with values a, b, c, d, e, \dots , and can then represent relation r by a separate table:



In the OO representation each instance is an object, identified with an oid, say with values $\alpha, \beta, \gamma, \delta, \epsilon, \dots$, and the programmer may and must use these oid's to refer to the objects:



This representation consists of pairs (object id, object value), and since the object id's are really *keys*, this representation may also be called: **key-value store**. Each MapReduce computation is based on a key-value store; remember from Dean [1]:

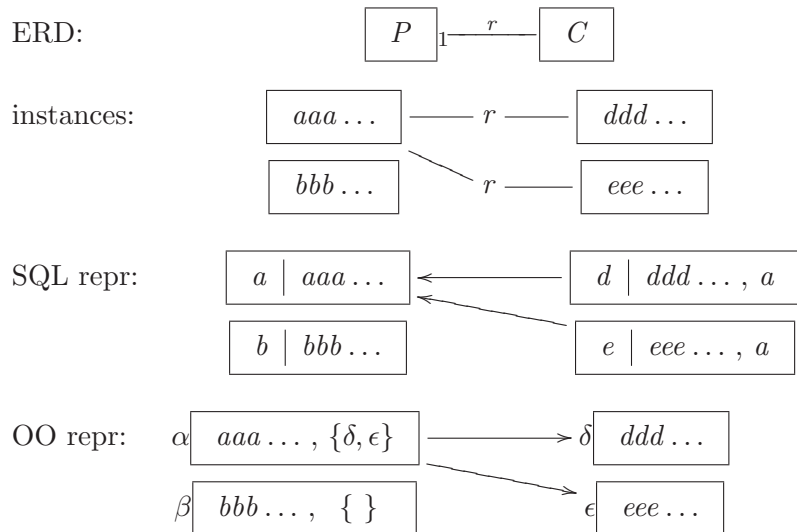
$$MapReduce_{(map, reduce, combine)} : [KEY \times VALUE] \rightarrow [KEY' \times VALUE']$$

We will continue to use the term OO representation, rather than key-value store.

A special case. Meijer discusses *only* the following special case:

Relation r has for each instance $c \in C$ precisely one instance $p \in P$.

(Now, r is even a function from C to P , and P may be interpreted as *Parent* and C as *Child*.) Relation r can be represented in the SQL world by one key-valued attribute in the C rows and no such things in the P rows, whereas it can be represented in the OO style by one set-valued attribute of oid's in the P objects and no such things in the C objects:



Observations and claims

In both the general and the special case, one may observe various “dualities”:

1 Extensional vs intensional identities. In the SQL representation, the identity of an instance (the key) is present in its row representation (“extensional identity”), whereas in the OO representation the identity (the oid) is absent in the object representation (“intensional identity”).

2 High vs low retrieval cost. In the SQL representation, the retrieval of rows r -related to the row with key value a , requires an expensive search (a scan of the table) whereas in the OO representation the retrieval of objects r -related to the object with oid α , is cheap (a simple dereference). See also §6.

3 Typedness vs untypedness. The presence of key values that can be manipulated by the SQL programmer and that play a role in the integrity constraints (see §9), forces rows to be typed, at least to the degree that key attributes have a type. The fact that oid values cannot be manipulated by the OO programmer, allows objects to be untyped, at least regarding the oid values.

4 Representation of relation r . (This is my own observation about a nice symmetry, not mentioned by Meijer.) Look at the pictures of the general case. In the SQL representation, relation r is represented in a *centralized* way as a separate table containing all pairs (x, y) from r . In the OO representation, relation r is represented in a *distributed* way: a P object with oid x contains $\{y \mid (x, y) \in r\}$ and an C object with oid y contains $\{x \mid (x, y) \in r\}$.

Now look at the special case. In the SQL representation, a C row with key value y contains (the unique member of) $\{x \mid (x, y) \in r\}$. In the OO representation, a P object with oid x contains $\{y \mid (x, y) \in r\}$.

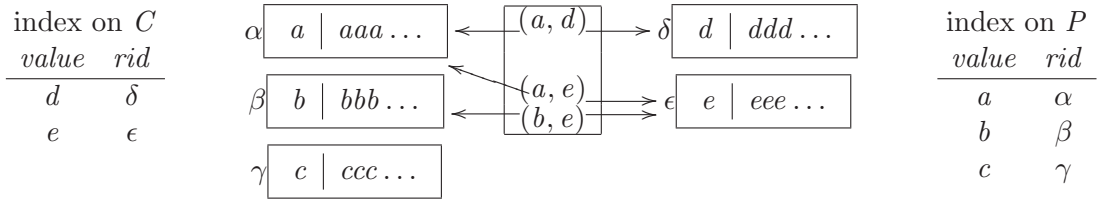
5 Categorical duality. Look at the pictures of the SQL and OO representation for the special case: disregarding the key values and object id’s, the pictures are the same except that the arrows are reversed. This fits nicely in the notion of *duality* as known in category theory. In terms of category theory, the two representations are the following dual categories:

- The SQL representation is the category having “entity instances with keys” as objects and “foreign keys determined by relation r (used from C to P)” as arrows.
- The OO representation is the category having “entity instances with oid’s” as objects and “object references determined by relation r (used from P to C)” as arrows.

If in the ERD instantiation all of aaa, \dots, eee are flat scalar values, then the OO representation is —according to Meijer— just the Amazon SimpleDB data model. If in the ERD instantiation all of aaa, \dots, eee are blobs, then —according to Meijer, but I have my doubts— the OO representation is just the HTML5 key-value storage model. Thus, the dual of the traditional SQL representation turns out to be practically useful.

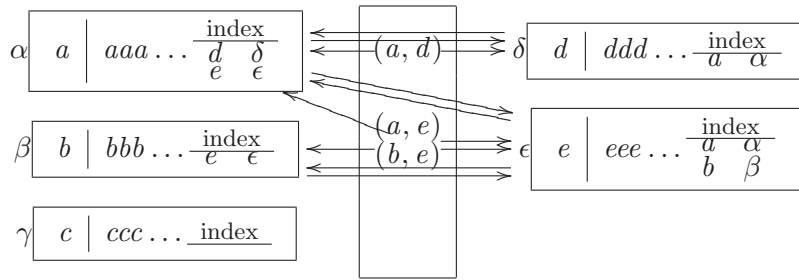
Many concepts from mathematics, physics, and computer science are related to each other by categorical duality. Almost always, dualization of a useful concept yields a useful concept again. The appendix (page 7) illustrates this phenomenon.

6 Indexing. In §2 we have observed that in the SQL representation the retrieval of all rows with a given key value is expensive, whereas in the OO representation cheap dereferences of oid's suffice. To “repair” this sad affairs in the SQL representation, the SQL programmer may create an index (and for *primary* keys this is often done automatically by the DBMS):

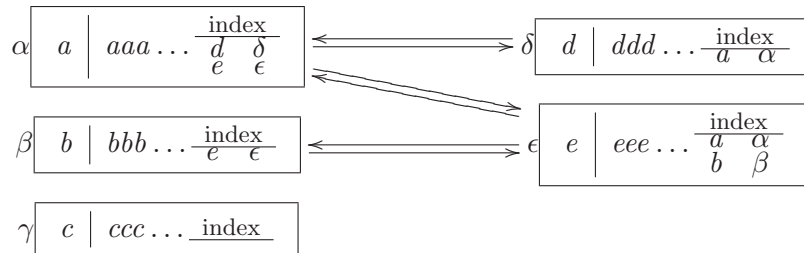


Each index gives, for each value v the row id (denoted with α, β, \dots) of the row having that value v . A row id provides direct access to a row; it is semantically *the same* as an object id. Row id's are not expressible by the SQL programmer, but are exploited in query executions.

If we draw the lines from each applied occurrence of a row id to its unique defining occurrence, we see that the index actually restores the information that is already present in the OO representation! This is even more clear if we replicate the relevant part of the index in each row:



The drawing is a little clearer if we omit the middle part (the link table):



7 Compositionality. Consider the query that yields “all combinations of P instances and C instances that are related via relation r ”. In the SQL representation, the result looks like this:

<i>a</i> <i>aaa</i> ...	<i>d</i> <i>ddd</i> ...
<i>a</i> <i>aaa</i> ...	<i>e</i> <i>eee</i> ...
<i>b</i> <i>bbb</i> ...	<i>e</i> <i>eee</i> ...

It turns out that the result of the query is not normalized: both *aaa* and *eee* is duplicated. The result itself is not an acceptable representation. Thus we see that SQL manipulations may lead us outside the SQL framework: SQL is not compositional. In the OO representation, the query result consists of the collection of objects identified by $\alpha, \beta, \delta, \epsilon$:

$$\begin{array}{cc} \alpha \boxed{a \mid aaa \dots, \{\delta, \epsilon\}} & \delta \boxed{d \mid ddd \dots, \{\alpha\}} \\ \beta \boxed{b \mid bbb \dots, \{\epsilon\}} & \epsilon \boxed{e \mid eee \dots, \{\alpha, \beta\}} \end{array}$$

There is full sharing and no duplication: each object “contains” other objects by referencing them by their oid’s.

8 Comprehension. I assume that list comprehension, filter $p \triangleleft$, map f^* , and reduce $\oplus /$ are well-known from functional programming. List comprehensions can be easily expressed in maps, reduces, and filters:

$$\begin{aligned} [f x \mid x \leftarrow xs; p x] &= f^* \cdot p \triangleleft \cdot xs \\ [g x \mid x \leftarrow xs; p x; y \leftarrow f x; q y] &= g^* \cdot q \triangleleft \cdot \# / \cdot f^* \cdot p \triangleleft \cdot xs \\ [h x \mid x \leftarrow xs; p x; y \leftarrow f x; q y; z \leftarrow g y; r z] &= h^* \cdot r \triangleleft \cdot \# / \cdot g^* \cdot q \triangleleft \cdot \# / \cdot f^* \cdot p \triangleleft \cdot xs \end{aligned}$$

In fact, SQL’s select-from-where expression is a simple comprehension in disguise:

$$\text{select } expr \text{ from } R r, S s, T t \text{ where } cond = [expr \mid (r, s, t) \leftarrow R \times S \times T; cond]$$

This can also be done for the group-by construct [2]. Since filter itself can be expressed in maps and reduces, and since maps and reduces exist for a large class of inductively defined datatypes [5], it follows that *comprehension* exists for a large class of inductively defined datatypes. In particular, select-from-where expressions or more complicated comprehensions can also be used for the OO representation: no new query languages or query concepts have to be invented.

9 ACID vs BASE. In the SQL representation, the programmer can manipulate (create, change) keys, whereas in the OO representation, the programmer cannot manipulate (create, change) oid’s. As a consequence, SQL manipulations might violate referential integrity (which is the property that *key values are unique in a table* and *foreign key values do occur in the referenced table as key values*), whereas OO manipulations cannot introduce non-existent objects. Thus, in the SQL representation, manipulations must be checked (by the consistency checks of transactions updating *multiple* rows as *one atomic* action), whereas in the OO representation, manipulations can be done for individual objects separately, and the representation is consistent only eventually after all individual updates. In other words, the SQL representation calls for an ACID environment whereas the OO representation calls for a BASE environment. (BASE abbreviates *Basically Available, Soft-state, Eventually consistent*. The acronym is a bit contrived —and so is ACID— but has a nice symmetry with ACID).

Notice that in the special case, in the SQL representation *deletion* of C rows cannot cause dangling references whereas in the OO representation this holds for deletion of P objects.

10 Closed vs open world assumption. Since in the SQL representation multiple changes are done and checked in atomic transactions, see §9, a programmer using the SQL representation can safely adopt the closed world assumption (“if a fact is not represented, then it is not true”). Since in the OO representation consistency might be realized only eventually, see §9, a programmer using the OO representation had better adopt the open world assumption (“even if a fact is not represented, it might be true — and be represented later”).

Adoption of the open world assumption severely limits the interpretation of the answers to some queries.

11 Scalability and distribution. Typically, but not necessarily, the SQL representation is used in an ACID environment (see §9) and, because referential integrity checking would be rather infeasible for distributed data, the data is *not* distributed. It follows that in such an environment query optimization can use all statistics of the data, and users can focus on the *what* rather than the *how*. Typically, but not necessarily, the OO representation is used in a BASE environment (see §9) and, because validity checking of object references is relaxed, the data can be and *is* distributed. (Since object references might be invalid, the programmer must be prepared to get a “not found - 404” error in query answers.) It follows that in such an environment query optimization is hard to automate, and users will generally rely on patterns like MapReduce that can be efficiently executed on the distributed data.

12 Object-Relational mapping. Meijer mentions the concept of Object-Relational mapping, and gives an example with LINQ syntax. He observes that, for a given ERD instantiation, the OO representation needed for the OR mapping (in order to produce a correct SQL representation) is more complicated than the direct OO representation of the ERD instantiation. (Alas, I see no duality here... or should we think of a Relation-to-Object mapping?)

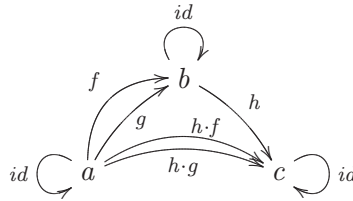
References

- [1] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150. USENIX, December 2004. Google Research Paper, <http://labs.google.com/papers/mapreduce.html>. Also: *Comm. of the ACM*, Jan 2008, Vol. 5, nr 1, pp. 107–113.
- [2] Maarten M. Fokkinga. Constructing SQL queries. Unpublished Technical Report. Obtainable from <http://www.cs.utwente.nl/~fokkinga/mmf2004i.pdf>, 2004.
- [3] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992. Obtainable by <http://www.cs.utwente.nl/~fokkinga/mmfphd.pdf>.
- [4] M.M. Fokkinga. Abstracte datatypen en categorie-theorie. Memoranda Informatica 94-32, University of Twente, June 1994. Obtainable by <http://www.cs.utwente.nl/~fokkinga/mmf94e.ps>.
- [5] M.M. Fokkinga. Background info for Map and Reduce. Unpublished Technical Report. Obtainable from <http://www.cs.utwente.nl/~fokkinga/mmf2009n.pdf>, 2009.
- [6] Erik Meijer and Gavin Bierman. A Co-Relational Model of Data for large Shared Data Banks. *Comm. of the ACM*, 54(4):49–58, April 2011. doi:10.1145/1924421.1924436.
- [7] Wikipedia. Category theory — wikipedia, the free encyclopedia, 2011. http://en.wikipedia.org/w/index.php?title=Category_theory&oldid=467007421; accessed 30-Dec-2011.

Following pages: Explanation of categorical duality.

Appendix – Explanation of categorical duality

Category. A category consists of *objects* and *arrows* (sometimes called *morphisms*), together with a binary operation on arrows, called *composition* (denoted as an infix dot: $-\cdot-$). Moreover, these objects, arrows and composition must fulfil some properties, called *axioms*. One of these axioms is that each object a has an arrow $id_a : a \rightarrow a$ that, under composition, behaves as an identity. The other axioms are so natural that we don't explicate them here. Here is a little category with objects a, b, c and eight arrows (the three identities, and f, g, h , and all compositions of these; note that $f \cdot id = f$ and so on):



All categorical theory is expressed in terms of objects, arrows, and composition. However, the interpretation is up to the user. Every property of the intended interpretation that the user has in mind, can only be used and exploited in the categorical theory if the property can be expressed in terms of the objects, arrows and composition. To do so may take some ingenuity; for example, how would you express categorically the concepts of “maximum of two numbers” or “Cartesian product of two sets” or “conjunction of two propositions”? We shall do it in a moment.

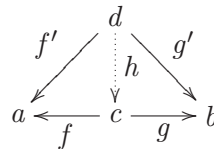
Example categories. Here are a few examples of mathematical concepts that can be represented categorically. First, graphs can be described categorically by representing nodes as objects, paths as arrows, and path concatenation as arrow composition. Second, some aspects of set theory can be described categorically by representing sets as objects, functions as arrows, and function composition as arrow composition. Third, some aspects of number theory can be described categorically by representing numbers as objects, pairs (m, n) with $m \leq n$ as arrows, and transitivity of \leq as arrow composition; (in this category there is exactly one arrow between two numbers m and n , so that in a picture the arrows can be left unnamed). Fourth, some aspects of propositional logic can be described categorically by representing propositions as object, implications as arrows, and the rule of transitivity as arrow composition; (in this category there is exactly one arrow between two propositions, so that in a picture the arrows can be left unnamed). What's the usefulness of all this? Well, there might be some categorical concepts and theorems that, interpreted in these examples, might give useful concepts and theorems. We shall see so in a moment.

A little theory. Here is a little example of some category theory: two definitions and three theorems. First, define that two objects a and b are *isomorphic* if:- there exist arrows $\phi : a \rightarrow b$ and $\psi : b \rightarrow a$ such that $\phi \cdot \psi = id_b$ and $\psi \cdot \phi = id_a$. We then have the following meta-theorem: suppose that a and b are isomorphic via ϕ and ψ ; then all definitions and theorems of category theory remain valid and true, respectively, if occurrences of $\xrightarrow{f} a \xrightarrow{g}$ are replaced by $\xrightarrow{\phi \cdot f} b \xrightarrow{g \cdot \psi}$. Thus, isomorphic objects cannot be distinguished categorically. For example, in the above mentioned category of sets, all sets with the same cardinality are isomorphic. If you *do* want to distinguish equally sized sets, then you should not consider the above category of

sets but another one which better reflects your intention. The other definition and theorems read as follows. Consider three objects a, b, c and two arrows $a \xleftarrow{f} c \xrightarrow{g} b$. We define that c with f, g is a “product of a and b ” if:-

for every $a \xleftarrow{f'} d \xrightarrow{g'} b$, there exists precisely one $h : d \rightarrow c$ with $f' = f \cdot h$ and $g' = g \cdot h$.

The objects and arrows involved in this definition may be drawn as follows:

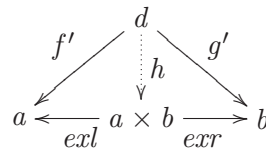


the required equalities are:

$$\begin{aligned}
 f' &= f \cdot h \\
 g' &= g \cdot h
 \end{aligned}$$

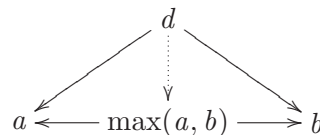
As a theorem one can now (easily) prove that if c_1 with f_1, g_1 is also a product of a and b , then c and c_1 are isomorphic; so we can speak of “the” product rather than “a” product. Also, one can (easily) prove that “product” is associative: the product of a and the product-of- b -and- c is isomorphic to the product of the product-of- a -and- b and c .

Interpretation. Interpreted in classical set theory, ‘the categorical product of sets a and b ’ is the “Cartesian product of a and b , with the accompanying projection functions”:



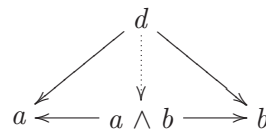
exl extracts the left component
exr extracts right component

Interpreted in the category that represents the ordering on natural numbers, ‘the categorical product of numbers a and b ’ is the “maximum of numbers a and b ”:



$x \downarrow y$ means $x \geq y$

Interpreted in the category of propositions, ‘the categorical product of propositions a and b ’ is the “conjunction of a and b ”:



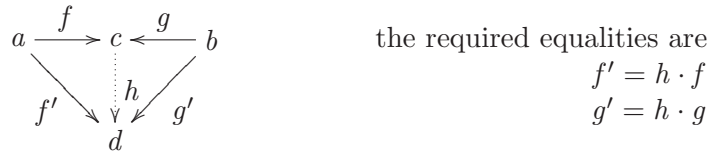
$x \downarrow y$ means $x \Rightarrow y$

Thus, many *similar* phenomena in different branches of mathematics turn out to have the *same* characterization in category theory. And... , the single categorical proof of the associativity of these concepts is valid for all three interpretations (*all* interpretations, indeed).

Dualization. *Dualization* is a syntactic manipulation on expressions of category theory; it systematically reverses the arrows. One can (easily) prove that dualization preserves well-formedness of definitions and the truth of categorical proofs/theorems. Dualization gives “two for the price of one”. The dual of a concept *xxx* is often called *co-xxx*.

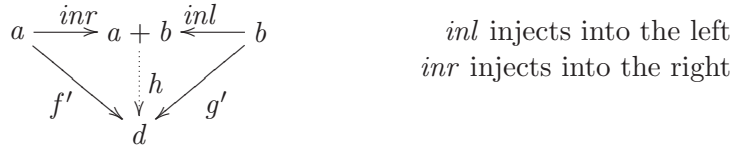
As an example we dualize the concept of “product of *a* and *b*” which we call “sum of *a* and *b*” (or alternatively: “co-product of *a* and *b*”). Consider three objects *a*, *b*, *c* and two arrows $a \xrightarrow{f} c \xleftarrow{g} b$. We define that *c* with *f*, *g* is a “sum of *a* and *b*” if:-

for every $a \xrightarrow{f'} d \xleftarrow{g'} b$, there exists precisely one $h : d \leftarrow c$ with $f' = h \cdot f$ and $g' = h \cdot g$:

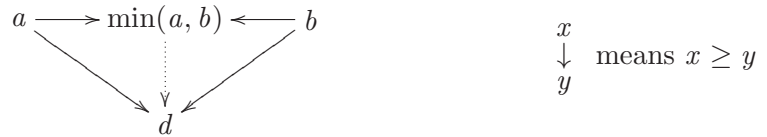


Because dualization preserves categorical proof, it *immediately* follows, without further proof, that different sums of two objects are isomorphic and we can speak of “the” sum rather than “a” sum, and it follows that sum is associative.

Interpretation. Interpreted in classical set theory, ‘the categorical sum of sets *a* and *b*’ is the “disjoint union of *a* and *b*, with the accompanying injection functions”:



Interpreted in the category that represents the ordering on natural numbers, ‘the categorical sum of numbers *a* and *b*’ is the “minimum of *a* and *b*”:



Interpreted in the category of propositions, ‘the categorical sum of propositions *a* and *b*’ is the “disjunction of *a* and *b*”:



These examples illustrate that dualization of useful concepts gives concepts that –often– are useful too. Dualizing SQL to coSQL is another example.

Further reading. Elsewhere [4] I give a more elaborate categorical treatment of Cartesian product and disjoint union, showing that the categorical definition “really abstracts from implementation”.

Appendix A of my PhD thesis [3] gives in seven pages a self-contained introduction with precise definitions of important notions of category theory, including functors, natural transformations, dualization, isomorphism. See also `Category_theory` in Wikipedia [7].