

# Category-Theoretic Consequences of Denotators as a Universal Data Format

Gérard Milmeister  
Institut für Informatik  
Universität Zürich  
milmei@ifi.unizh.ch

April 24, 2007

## Abstract

The RUBATO COMPOSER Music Software uses denotators as the universal data format. Denotators are objects (or points) in general spaces called forms. Denotators and forms constitute a general architecture of concepts, which itself is embedded in category theory. More precisely, the category in question is the category of set-valued presheaves over the modules. This embedding allows some new and important properties which are useful for modeling musical objects.

In this paper we discuss the consequences of using the limit and colimit constructions of category theory in the implementation of denotators in RUBATO COMPOSER.

## 1 Introduction

The RUBATO COMPOSER music software [4] uses the theory of forms and denotators [2] to model its universe of objects. Denotators play the role of objects and forms the role of data types in this scheme. In fact, the various kinds of constructions available with forms correspond mostly to the kinds traditionally found with structured data types. Apart from *simple* forms, which match the basic data types, such as numbers and strings, but comprise the whole theory of mathematical modules, there are *limit* and *colimit* forms, which correspond to product and disjoint union data types, such as they are found in complete analogy in functional programming languages, such as Haskell or SML. Additionally *powerset* forms describe sets of denotators. Such a set data type construction is usually not available natively in programming languages but is generally provided as part of the standard library (for example `Set` in Java).

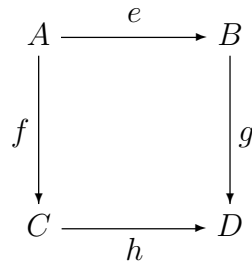
Thus, forms and denotators allow the usual data type constructions found in modern programming languages. There would be nothing especially interesting about this, except for the fact that the theory is embedded in category theory. This fact could already have been anticipated by the use of the category-theoretic terms *limit* and *colimit* instead of the more familiar *product* and *coproduct*.

Apart from the fact that denotators live in a functorial space based on the category of modules (to be exact: the set-valued presheaves over the modules), the main difference between the categorical view and the traditional conception is the presence of *diagrams*.

## 2 Diagrams in Category Theory

To keep things simple, concepts from category theory will be used henceforth in an intuitive way and in terms of forms and denotators. In the category of forms, the objects are the forms and morphisms (or arrows) are functions between forms (i.e., a function  $f : F \rightarrow G$  that maps a denotator in the form  $F$  to a denotator in the form  $G$ ).

A diagram in the category is a directed graph with forms as vertexes and functions as arrows. It may be required that any paths in the diagram commute. A very basic example shall illustrate the concept of a diagram:



In this diagram, there are four objects (forms)  $A, B, C$  and  $D$  and four morphisms (functions between forms)  $e : A \rightarrow B, f : A \rightarrow C, g : B \rightarrow D$  and  $h : C \rightarrow D$ . Commutativity of this diagram requires that following any of the paths (as the composition of functions) always leads to the same result.

With a little imagination it is obvious that such a diagram is nothing else than a graphical (and category-theoretical) expression for an equation! Indeed, the commutativity of the above diagram can be rendered as an equation (the left and the right side of the equation corresponding to the two possible paths in the diagram):

$$\forall x \in A : g(e(x)) = h(f(x))$$

## 3 Limits

The common Cartesian product  $P$  of  $n$  factors  $F_1, \dots, F_n$  is drawn as the diagram in figure 1 (left). The

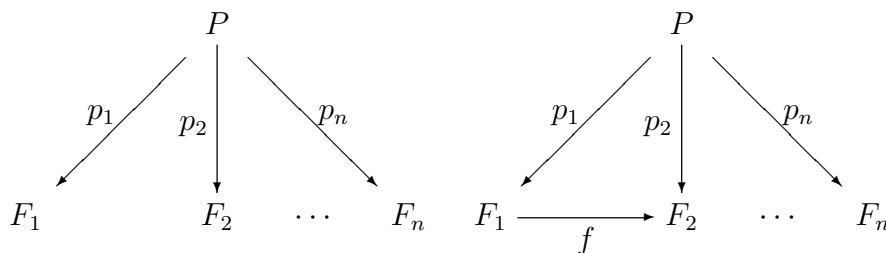


Figure 1: Product diagram (left), limit diagram (right).

arrows in such a product diagram are the projections  $p_i$  to the respective factors. There are *no* arrows between the factor objects, i.e., the diagrams consisting of the factors is discrete.

In the general case of a product, the limit, a diagram may be non-empty. A simple example is shown in figure 1 (right). Here the diagram features an arrow  $f$  from  $F_1$  to  $F_2$ . An object  $P$  is a solution whenever, for a given  $x \in P$ , we have

$$f(p_1(x)) = p_2(x)$$

or, in words, the second factor of  $x$  is equal to  $f$  applied to the first factor of  $x$ . More generally, such a diagram embodies *constraints* on the factors of traditional product.

**Example.** In music, triads provide a straightforward example where limits can be put to good use. A triad form can be defined<sup>1</sup> as

$$\begin{aligned} \textit{Triad} &: \textit{Limit}(\textit{Note}, \textit{Note}, \textit{Note}) \\ \textit{Note} &: \textit{Limit}(\textit{Onset}, \textit{Pitch}, \textit{Duration}, \textit{Loudness}) \end{aligned}$$

This simple definition does not guarantee that all notes have the *same* onset (a condition which is arguably sensible). This condition can be accomplished by extending the definition

$$\textit{Triad} : \textit{Limit}(\textit{Note}, \textit{Note}, \textit{Note}, \textit{Onset}).$$

and adding an arrow  $f : \textit{Note} \rightarrow \textit{Onset}$  from each of the three *Note* factors to the *Onset* factor and define  $f$  as the projection from a *Note* to its *Onset* factor.

It must be noted that diagrams will not provide solutions for every constraint programming problem that may ever arise. However, many standard situations can be modeled in a quite natural way. The next examples shall illustrate two particular cases.

The case shown above involves equations of the type  $f(X) = Y$  where  $X$  and  $Y$  are factors of the limit. A more general type has the form  $f(X, Y) = g(Z)$ . Two additional features are illustrated here: functions with more than one argument and multiple arrows. For the two-argument function  $f$  we need to add the product space  $X \times Y$  to factors of the limit. For the equality involving  $f : X \times Y \rightarrow W$  and  $g : Z \rightarrow W$  another factor  $Z$  is needed. With the addition of the projections to enforce the consistency between  $X$  and  $Y$  factors on the one hand and the  $X \times Y$  on the other, we have the following:

$$P : \textit{Limit}(X, Y, Z, X \times Y, W)$$

with the morphisms

$$\begin{aligned} f &: X \times Y \rightarrow W \\ g &: Z \rightarrow W \\ p_X &: X \times Y \rightarrow X \\ p_Y &: X \times Y \rightarrow Y. \end{aligned}$$

Until now, only equalities have been considered. One possibility to implement *inequalities* is by means of predicates. A predicate is a morphism of the type  $F \rightarrow \textit{Boolean}$ , where  $F$  is a form and *Boolean* is a predefined form for representing the two Boolean values *TRUE* and *FALSE*. An inequality  $X < Y$  will result in a morphism  $h : X \times Y \rightarrow \textit{Boolean}$ . Of course, a *Boolean* factor must be added to the factors of the limit as well.

An overview of some more complex applications of constraint programming in music composition and analysis is given in [5].

## 4 Colimits

To a certain extent the definitions of coproducts and colimits are analogous to the definitions of products and limits, respectively.

The coproduct  $C$  of  $n$  cofactors  $F_1, \dots, F_n$  is drawn as the diagram in figure 2 (left). The arrows in such a coproduct diagram are the injections  $i_j$  from the respective cofactors into the coproduct. There are *no* arrows between the cofactor objects, i.e., the diagrams consisting of the cofactors is discrete.

---

<sup>1</sup>The denoteX notation from [2] is used here.

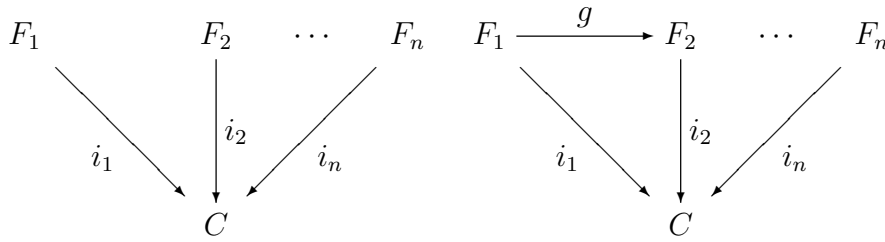


Figure 2: Coproduct diagram (left), colimit diagram (right).

An example of a colimit with a non-empty cofactor diagram is shown in figure 2 (right). The equation defined by this diagram is

$$i_1(x) = i_2(g(x))$$

for every  $x \in F_1$ . The interpretation of this condition is little less obvious than for the limit case. The equation says that the injection of  $x \in F_1$  into the colimit form  $C$  is equal to the injection  $g(x) \in F_2$  into  $C$ .

In more operational terms, consider a denotator  $c \in C$  which is the injection of a denotator  $x \in F_1$ . It now makes sense to *reinterpret*  $c$  by applying  $g$  to  $x$  and considering  $c$  as the injection of  $g(x)$ .

**Example.** Consider a general music object as an input to an analytic software component, which may be either a score, an harmonic analysis of a score, or a formal analysis of a score. The design of the analytic component and the exact shape of the score and its analyses are not important in this context. Such a form may be defined as follows:

$$\text{Analysis} : \text{Colimit}(\text{Score}, \text{HarmonicAnalysis}, \text{FormalAnalysis}).$$

If we define  $f : \text{Score} \rightarrow \text{HarmonicAnalysis}$  and  $g : \text{Score} \rightarrow \text{FormalAnalysis}$  as the functions which carry out the actual analyses and add them as arrows to the diagram of the colimit form *Analysis*, the analytic procedures are embedded into the form! Then, whenever a denotator of form *Analysis* is encountered and it contains a score, both analyses can be retrieved without further ado.

## 5 Integration in RUBATO COMPOSER

The implementation of forms and denotators RUBATO COMPOSER already supports diagrams for types limit and colimit. The implementation of the various morphisms between forms has not yet progressed very far, currently only morphisms of simple forms are available.

The consequences of having arbitrary diagrams have to be explored more deeply, since various interesting situations may arise. There are essentially two cases that may happen during the construction of a denotator of type limit.

First, all the factors required by the form are supplied. In this case the constructor must check all morphisms of diagrams for any inconsistency. If there is an inconsistency, the construction fails.

Second, only some of the factors are supplied. The constructors must then check the diagram for all the given factors, just as in the first case. In addition, the constructor must try to *infer* the missing factors from the diagram. This is far from obvious: If the diagram contains a loop on one of the factors, the solution for this factor would be a *fixed point* for the morphism associated with the loop. A diagram may contain one or more cycles. In either case, it may be impossible to find a solution (either positive or negative) computationally.

In the case where all morphisms are affine mappings, linear algebra provides the environment for solving such equations (see [3] for example).

For situations that are well known in constraint programming research, existing implementations (for example a Java library such as Choco [1]) should be used.

As shown above, adding constraints may be result in enriching the diagram of a limit with auxiliary factors. It may be convenient to hide these from the user. The Rubato framework needs to provide the tools for managing such forms of increasing complexity and expose only the essential information.

The colimit situation does not pose such difficulties, since the morphisms are only applied on demand.

Of course, it must be explored how to expose the effects of diagrams to the user of forms and denotators, especially in the context of graphical systems such as the RUBATO COMPOSER GUI. Situations such as the dynamic<sup>2</sup> failure of limit constructions must be made transparent and manageable. The graceful handling of constraints and their failures are part of the further research and development of the RUBATO COMPOSER framework.

## References

- [1] *Choco constraint programming system*. <<http://choco.sourceforge.net>>.
- [2] Mazzola, Guerino. *The Topos of Music*. Birkhäuser, Basel 2002.
- [3] Mazzola, Guerino and Andreatta, Moreno. “From a Categorical Point of View: K-nets as Limit Denotators.” In: *Perspectives of New Music*, Volume 44, Number 2, Princeton University Press, Princeton 2006.
- [4] Milmeister, Gérard. *The Rubato Composer Music Software: Component-Based Implementation of a Functorial Concept Architecture*, Zürich 2006.
- [5] Truchet, Charlotte. “Some Constraint Satisfaction Problems in Computer Assisted Composition and Analysis.” In: Guerino Mazzola, Thomas Noll, and Emilio Lluís-Puebla (eds), *Perspectives in Mathematical Music Theory*, pp. 330–342, epOs Music, Osnabrück 2004.

---

<sup>2</sup>Dynamic, since limit constructions may fail because of the *values* of the factors, and not only because of their *forms*.