A functional Approach for On Line Analytical Processing

Claudia Necco, Luis Quintas *
Instituto de Matemáticas y Física Aplicada (I.M.A.S.L.)
Universidad Nacional de San Luis
Ejército de los Andes 950
5700 San Luis - Argentina
ayesha@unsl.edu.ar

J. Nuno Oliveira
Departamento de Informática
Universidade do Minho
4700 Braga - Portugal

Abstract

This paper describes an approach to On Line Analytical Processing (OLAP), expresed in the declarative programming paradigm.

We define a collection of functions that capture some of the functionality currently provided by multidimensional database product. This is done by defining operations which allow for classifying and reducing relations (tables). Suitably combined, these operations will make possible to carry out the multidimensional analysis of a relational database, and make possible the declarative specification and optimization of multidimensional database queries.

The library works over an abstract model of the relational database calculus as defined by Maier, written in the style of model-oriented formal specification in the functional language Haskell (details can be found in [8].

Keywords: functional programming, declarative programming, relational data model, on line analytical processing, multidimensional analysis.

Resumen

Este paper describe una aproximación al Procesamiento Analítico On Line (OLAP), desarrollado en el paradigma de la programación declarativa.

Definimos un conjunto de funciones que capturan algunas de las funcionalidades actualmente provistas por las bases de datos multidimensionales existentes. Esto es realizado definiendo operaciones que permiten clasificar y reducir relaciones (tablas). Dichas operaciones, convenientemente combinadas, permitirán llevar a cabo el análisis multidimensional de una base de datos relacional, junto con la especificación declarativa y optimización de las correspondientes consultas multidimensionales sobre dicha base.

La biblioteca provista trabaja sobre un modelo abstracto del cálculo de base de datos relacionales tal como ha sido definido por Maier,escrito en un estilo de especificación formal orientada al modelo en el lenguaje funcional Haskell (más detalles pueden encontrarse en [8])

Palabras claves: Programación Funcional, Programación Declarativa, Modelo de Datos Relacional, Procesamiento Analítico On Line, Análisis Multidimensional.

*This work was carried out in the frame of the following Research Projects: 1- Program Understanding an Re-engineering: Calculi and Applications funded by the Portuguese Science and Technology Foundation (Grant POSI/CHS/44304/2005). 2- PROYECTO P-319002- Decisión Artificial, Uso de Autómatas en Problemas de Decisión. - I.M.A.S.L., U.N.S.L

1 INTRODUCTION

Codd proposed the concept of On-Line Analytical Processing (OLAP) for rendering enterprise data in multidimensional perspectives, performing on-line analysis of data using mathematical formulas or more sophisticated statistical analyses, and consolidating and summarizing data [3], [4].

OLAP call for sophisticated on-line analysis, something for which the traditional relational model [2] offers little support. Several vendors have already developed OLAP products, but many of these suffer from the following limitations: they do not support a comprehensive "query" language similar to SQL; viewing data in multi-dimensional perspectives involves treating certain attributes as *dimensional parameters* and the remaining ones as *measures*, and then analyzing them as a "function" of the parameters; and, finally, unlike for the relational model, there is no precise, commonly agreed, conceptual model for OLAP or the so-called multidimensional databases (MDD) (see [5], [1], [6]).

We present a comprehensive, simple conceptual model for OLAP that treat *dimensions* and *measures* symmetrically.

The structure of the paper as follows: Section 2 introduces elementary concepts and terminology which are used throughout the paper. Section 3 sketches a formal model for database relational data. Section 4 contains a very brief comparison between relational and multidimensional tables. Section 5 presents the functionality necessary for OLAP-based aplications. The last section present some conclusions and future work.

2 CATEGORICAL SUPPORT

Categories. A category consists of a collection of objects and a collection of arrows. Each arrow $f :: a \rightarrow b$ has a source object a and a target object b. Two arrows f and g can be composed to form a new arrow $g \cdot f$, if f has the same target object as the source object of g. This composition operation is associative. Furthermore, for each object a there is a so-called identity arrow $id_a :: a \rightarrow a$, which is the unit of composition.

Our base category is called $Types$ and has types as objects and functions as arrows. Arrow composition is function composition (.) and the identity arrows are represented by the polymorphic function id.

Functors. Functors are structure-preserving mappings between categories. *Polymorphic* datatypes are functors from $Types$ to $Types$. In Haskell, functors can be defined by a type constructor f of kind $* \rightarrow *$, mapping objects to objects, together with a higher-order function $fmap$, mapping arrows to arrows. This is provided as a constructor *class* in the Haskell *Prelude* (the standard file of primitive functions) as follows:

$$
\begin{array}{l}
class\ Functor\ f\ where \\
\quad fmap :: (a \rightarrow b) \rightarrow (f\ a \rightarrow f\ b)
\end{array}
$$

The arrow action of a functor must preserve identity arrows and distribute over arrow composition. For functors from $Types$ to $Types$, this means that the following equations must hold:

$$fmap\ id = id$$

$$fmap\ (f \cdot g) = (fmap\ f) \cdot (fmap\ g)$$

Bifunctors. The *product category* $Types \times Types$ consists of pairs of types and pairs of functions. We can define functors from $Types \times Types$ to the base category $Types$ in Haskell. These functors are called *bifunctors*. A (curried) bifunctor in Haskell is a type constructor of kind $* \rightarrow * \rightarrow *$, together with a function $bmap$. The following constructor class *Bifunctor* was made available:

$$\begin{array}{l} class\ Bifunctor\ f\ where \\ \quad bmap :: (a \rightarrow\ c) \rightarrow (b \rightarrow d) \rightarrow (f\ a\ b \rightarrow f\ c\ d) \end{array}$$

Products. Categorical *products* are provided in Haskell by the type constructor for pairs (a, b) (usually written as Cartesian product $a \times b$ in mathematics) and projections fst and snd (resp. π_1 and π_2 in standard mathematical notation). Type constructor $(,)$ is extended to a bifunctor in the obvious way:

$$\begin{array}{l} instance\ BiFunctor\ (\ ,\ )\ where \\ \quad bmap\ f\ g = f \times g \end{array}$$

where

$$\begin{array}{l} (\times) :: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow (a, c) \rightarrow (b, d) \\ (f \times g) = split\ (f \cdot fst)\ (g \cdot snd) \end{array}$$

and combinator $split :: (a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c)$ behaves as follows: $split\ f\ g\ x = (f\ x, g\ x)$.

Sums. Categorical *sums* are defined in the Haskell *Prelude* by means of type constructor

$$data\ Either\ a\ b = Left\ a \mid Right\ b$$

together with a function $either :: (a \rightarrow b) \rightarrow (c \rightarrow b) \rightarrow Either\ a\ c \rightarrow b$ satisfying the following equations:

$$\begin{array}{l} (either\ f\ g) \cdot Left\ = f \\ (either\ f\ g) \cdot Right = g \end{array}$$

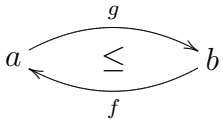
Type constructor *Either* is extended to a bifunctor by providing the following instance of *bmap*:

$$\begin{array}{l} (+) :: (a \rightarrow b) \rightarrow (c \rightarrow d) \rightarrow Either\ a\ c \rightarrow Either\ b\ d \\ (f + g)\ (Left\ a) = Left\ (f\ a) \\ (f + g)\ (Right\ b) = Right\ (g\ b) \end{array}$$

$$\begin{array}{l} instance\ BiFunctor\ Either\ where \\ \quad bmap\ f\ g = f + g \end{array}$$

The popular notations $\langle f, g \rangle$, $[f, g]$ and $\mathsf{F}\ f$ (where F is a functor) will be adopted interchangeably with $split\ f\ g$, $either\ f\ g$ and $fmap\ f$, respectively.

Invertible arrows. An arrow $f :: b \rightarrow a$ is said to be *right-invertible* (vulg. surjective) if there exists some $g :: a \rightarrow b$ such that $f \cdot g = id_a$. Dually, g is said to be *left-invertible* (vulg. injective) if there exists some f such that the same fact holds. Then type b is said to "represent" type a and we draw:



where g and f are called resp. the *representation* and *abstraction* functions. An isomorphism $f :: b \rightarrow a$ is an arrow which has both a right-inverse g and a left-inverse h — a *bijection* in set theory terminology. It is easy to show that $g = h = f^{-1}$. Type a is said to be *isomorphic* to b and one writes $a \cong b$.

Isomorphisms are very important functions because they convert data from one "format" to another format losing information. These formats contain the same "amount" of information, although the same datum adopts a different "shape" in each of them. Many isomorphisms useful in data manipulation can be defined [10], for instance function $swap :: (a, b) \rightarrow (b, a)$ which is defined by $swap = \langle \pi_2, \pi_1 \rangle$ and establishes the *commutative property* of product, $a \times b \cong b \times a$.

3 MODELING RELATIONAL DATA

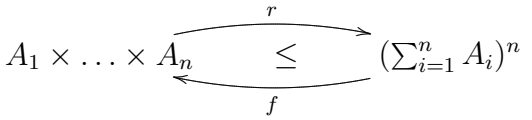
Collective datatypes. Our model of relational data will be based on several families of abstractions, including collective datatypes such as finite *powersets* ($\mathcal{P}a$) and finite partial *mappings* ($a \rightharpoonup b$). These are modeled as Haskell *polymorphic algebraic types* (that is, algebraic type definitions with type variables) based on finite lists $[a]$, see $Set\ a$ and $Pfun\ a\ b$ in Table 1, respectively. Both abstractions contain an equality relation and an ordering relation. The latter instantiates to set inclusion ($\subseteq$) and partial function definedness, respectively.

The finite sets model assumes invariant $\phi\ (Set\ l) \stackrel{\text{def}}{=} length\ l = card(elems\ l)$, where $length$ is Haskell standard and $card$(inal) and $elem$(ent)s have the usual set-theoretical meaning. Partial mappings require an extra invariant ensuring a functional dependence on sets of pairs [1]:

$$fdp \stackrel{\text{def}}{=} (\subseteq \{1\}) \cdot rng \cdot (id \rightharpoonup card) \cdot collect \tag{1}$$

Table 1 summarizes the Haskell modules defined for these datatypes.

Relational Database Model. An n-ary relation in mathematics is a subset of a finite n-ary product $A_1 \times \ldots \times A_n$, which is inhabited by n-ary vectors $\langle a_1, \ldots, a_n \rangle$. Each entry a_i in vector $t = \langle a_1, \ldots, a_n \rangle$ is accessed by its position's projection $\pi_i : A_1 \times \ldots \times A_n \rightarrow A_i$. This, however, is not expressive enough to model relational data as this is understood in database theory [7]. Two ingredients must be added, whereby *vectors* give place to *tuples*: attribute names and NULL values. Concerning the former, one starts by rendering vectorial indices explicit, in the sense of writing *e.g.* $t\ \ i$ instead of $\pi_i\ \ t$. This implies merging all datatypes A_1 to A_n into a single coproduct type $A = \sum_{i=1}^{n} A_i$ and then represent the n-ary product as :



[1] $collect :: \mathcal{P}(a \times b) \rightarrow (a \rightharpoonup \mathcal{P}b)$ converts a relation into a set-valued partial function and $rng :: (a \rightharpoonup b) \rightarrow \mathcal{P}b$ is the usual *range* function.

	Finite Sets	**Partial Functions**
Datatypes:	$data\ Set\ a = Set\ [a]$	$data\ Pfun\ a\ b = Map[(a,b)]$
Constructors:	$emptyS, sings, puts, prods$ $ltos$	$bottom, singpf, putpf$ $collect$
Deletions:	$gets$	$getpf$
Observers:	$ins, nins, incls, card$ $allS$	$compatible, incompatible$ $allPf$
Filters:	$filterS$	
Operations:	$inters, unions, diffs, plus, pfzip$ $flatr, flatl, slstr, srstr, sextl, sextr$ $zipS, zipWithallS$	$plus, pfinv, restn, restp$ $pfzip, pfzipWith$
Folds:	$foldS$	$foldPf$
Functor:	$fmapS$	
Bifunctor:		$bmapPf$
Others:	$the, stol, elems, card$ $unzipS$	$dom, rng, aplpf$ $tnest, discollect, mkr, bpfTrue, bpfFalse$ $pfunzip$

Table 1: Finite sets and partial functions: datatypes and functions implemented.

under representation function [2] $r\ \ \langle a_j \rangle_{j=1..n} \stackrel{\text{def}}{=} \lambda j.(i_j\ a_j)$ which entails invariant

$$\phi\ \ t\ \ \stackrel{\text{def}}{=}\ \ \forall j = 1, \ldots, n, t\ \ j = i_j\ \ x\ :\ x \in A_j$$

Note that $j = 1, \ldots, n$ can be written $j \in \overline{n}$, where $\overline{n} = \{1, \ldots, n\}$ is the initial segment of the natural numbers induced by n. Set $\overline{n}$ is regarded as the *attribute* name-space of the model [3].

As a second step in the extension of vectors to tuples, we consider the fact that some attributes may not be present in a particular tuple, that is, NULL values are allowed [4]:

$$(\sum_{i \in \overline{n}} A_i + 1)^n$$

which finally leads to tuples as inhabitants of

$$Tuple\ \ =\ \ (\overline{n} \rightharpoonup \sum_{i \in \overline{n}} A_i)$$

thanks to isomorphism $A \rightharpoonup B\ \ \cong\ \ (B+1)^A$ [9]. This models tuples of arbitrary arity (up to n attributes), including the empty tuple. For notation economy, for every $X \subseteq \overline{n}$, we will write $Tuple_X$ as a shorthand for $X \rightharpoonup \sum_{i \in X} A_i$.

$Tuple$ is the basis for the Haskell model of database relations presented in Table 2. Relations ($Relation$) are sets of tuples sharing a common attribute schema ($SchemaR$). A rather complex invariant ensuring that tuples are well and consistently typed is required, which is omitted here for economy of presentation. This and other details of this model can be found in [8].

[2] Injections $i_{j=1,n}$ are associated to the n-ary coproduct. $Left$ and $Right$ in Haskell correspond to i_1 and i_2, respectively.

[3] The fact that this can be replaced by any isomorphic collection of attribute names of cardinality n has little impact in the modelling, so we stick to $\overline{n}$.

[4] Think of 1 as the singleton type $\{\text{NULL}\}$.

	Relations
Datatypes:	*type Tuple = Pfun IdAttr Value* *type SchemaR = Pfun IdAttr AttrInfo* *type IdAttr = String* *type Tuples = Set Tuple* *data Relation = Rel { schema::SchemaR, tuples::Tuples}* *data AttrInfo = InfA { ifKey::Bool, defaultV::Value }* *data Value = Int Int | String String | Date String | Time String*
Constructors:	$emptyR$
Operations:	$unionR, interR, diffR$ $projectR, selectR, natjoinR, equijoinR, renameR, divideR$

Table 2: Relations: datatypes and functions implemented.

4 RELATIONAL VERSUS MULTIDIMENSIONAL TABLES

The fundamental data structure of a multidimensional database is what we call an *n-dimensional table*. Let us start by giving some intuition behind the concept. We wish to be able to see values of certain attributes as "functions" of others, in whichever way suits us, exploiting possibilities of multi-dimensional rendering. Drawing on the terminology of statistical databases [11], we can classify the attribute set associated with the scheme of a table into two kinds: *parameters* and *measures*. There is no a priory distinction between parameters and measures, so that any attribute can play either role [5]. An example of a two-dimensional table is given in Table 3 (adapted from [5]).

SALES			*TIME*						
			Year	1996			1997		
			Month	Jan	Feb	...	Jan	Feb	...
CATEGORY	*Part*	*City*	*(Cost, Sale)*						
	PC	Mendoza		(5, 6)	(5, 7)	...	(4, 6)	(4, 8)	...
		Córdoba		(5, 7)	(5,8)	...	(4, 8)	(4, 9)	...
		⋮		⋮	⋮		⋮	⋮	
	Inkjet	Mendoza		(7, 8)	(7, 9)	...	(6, 9)	(6, 8)	...
		Bs. As.		(6, 9)	(6, 9)	...	(5, 8)	(5, 9)	...
		⋮		⋮	⋮		⋮	⋮	
	⋮	⋮		⋮	⋮		⋮	⋮	

Table 3: $SALES$ — a sample two dimensional table with dimensions $Category$ and $Time$. The associated parameter sets are $\{Part, City\}$ and $\{Year, Month\}$, respectively. The measure attributes are $Cost$ and $Sale$.

We want to work with the relational model we have defined in the previous section. A natural way to achieve this is to regard the multidimensionality of tables as an inherently *structural* feature, which is most significant when the table is rendered to the user. The actual *contents* of a table are essentially orthogonal to the associated structure, i.e., the distribution of attributes over dimensions and measures. Separating both features leads to a *relational* view of a table. For instance, the entry in the first (i.e., top left-most) "cell" in Table 3 containing the entry $(5, 6)$ corresponds to the tuple

[5] Needless to say, the data type of a measure attribute must have some kind of metrics or algebra associated with it.

$(PC, Mendoza, 1996, Jan, 5, 6)$ over the scheme

$$\{Part, City, Year, Month, Cost, Sale\} \tag{2}$$

in a relational view of table $SALES$.

To provide for OLAP, we need to define operations concerned with the following kinds of functionality:

- *Classification:* Ability to classify or group data sets in a manner appropriate for subsequent summarization.
- *Reduction/Consolidation:* Generalization of the aggregate operators in standard SQL. In general, reduction maps multi-sets of values of a numeric type to a single, "consolidated", value.

Classification is a generalization of the familiar SQL **group by** operator. The following example presents a typical query involving classification.

Example 4.1 *Consider the relation $RSALES$ with scheme (2) mentioned before. A typical query would be:* "find, for each part, the total amount of annual sales". *Even though this query involves* aggregation*, notice that it also involves* classifying *the data into various groups according to certain criteria, before aggregation is applied. Concretely, the above query involves classification by attributes $Part$ and $Year$.*

5 OLAP-BASED APPLICATIONS FRAME

5.1 Classification

In our model, relations are sets of tuples (tables) with a scheme, while tuples are finite partial functions. First, we define a function for partial function decomposition (or "tuple classification"). Then we extend the notion of classification, applying it in the context of tables.

Partial function decomposition (Classification on Tuples) Let t be a tuple ($t \in Attribute \rightharpoonup Value$), and let $X = \{A_1, ..., A_k\}$ be an arbitrary subset of $dom(t)$. A *classification* over X of tuple t, is the pair of tuples defined by $tnest\ X$, where $tnest$ is polymorphic function

$$tnest \quad : \quad \mathcal{P}A \to (A \rightharpoonup B) \to ((A \rightharpoonup B) \times (A \rightharpoonup B)) \tag{3}$$

$$tnest\ \ s \quad \stackrel{\text{def}}{=} \quad \langle \mid s, \setminus s \rangle \tag{4}$$

The idea of this function is to decompose a partial map into a pair of maps of the same type

$$A \rightharpoonup B \underset{\dagger}{\overset{tnest\ \ s}{\leq}} (A \rightharpoonup B) \times (A \rightharpoonup B)$$

which, together, rebuild the original map. In Haskell:

```
tnest :: Eq a => Set a -> Pfun a b -> (Pfun a b,Pfun a b)
tnest s f = (s <: f,s <-: f)
```

Tabular Decomposition (Classification on Tables) Let t be a set of tuples (typed $\mathcal{P}(Attribute \rightharpoonup Value)$), and let $X = \{A_1, ..., A_k\}$ be an arbitrary set of attributes of t. A $classification$ over X, of table t, is given by $tcollect\ X\ t$, where $tcollect$ is polymorphic function

$$tcollect\ \ s \ \stackrel{\text{def}}{=}\ \ collect \cdot \mathcal{P}(tnest\ \ s)$$

that is,

```
tcollect :: (Eq a, Eq b)
            => Set a
            -> Set (Pfun a b)
            -> Pfun (Pfun a b) (Set (Pfun a b))
tcollect s t = collect (nmap (tnest s) t)
```

in Haskell.

Classification essentially maps tuples of a relation to different groups (necessarily disjoint). Intuitively, we can think of the attributes in the first argument of $tcollect$ as corresponding to the "group id".

Example 5.1 *The classification part of the query of Example 4.1 can be expressed as follows:* tcollect {"Part","Year"} (tuples RSALES). *Table 4 illustrates the result of this operation in concrete Haskell syntax.*

5.2 Reduction/Consolidation

Next, we consider reduction/consolidation, which includes not only applications of functions such as *max, min, avg, sum, count* to multi-sets of values defined by groups of tuples, but also statistical functions such as *variance* and *mode*, and business calculations such as *proportions* and *quarterlies*.

In our model, reduction functions map sets of tuples of values to individual values. We first define some necessary auxiliary functions.

5.2.1 Relational Reduction

Function

$$\begin{aligned} tot2 \quad &: \quad (A \times B \to B) \to B \to \mathcal{P}(C \times A) \to B \\ tot2\ \ f\ \ u \ &\stackrel{\text{def}}{=}\ \ \{\!|[\underline{u}, f \cdot (\pi_2 \times id)]|\!\} \end{aligned}$$

reduces a binary relation on the second projection according to a reduction structure $A \times B + 1 \xrightarrow{\langle f, \underline{u} \rangle} B$ which, in most cases, is a monoid algebra. In Haskell:

```
tot2 :: (a -> b -> b) -> b -> Set (c,a) -> b
tot2 f = foldS (curry (uncurry f . (p2 >< id)))
```

```
Map[   ( Map [(”Part”, ”PC”),(”Year”, ”1996”)] ,
       Set[ Map[(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 5), (”Sale”, 6)],
            Map [(”City”, ”Mendoza”),(”Month”, ”Feb”),(”Cost”, 5), (”Sale”, 7)],
            ....................................................................,
            Map [(”City”, ”Cordoba”),(”Month”, ”Jan”),(”Cost”, 5), (”Sale”, 7)],
            ....................................................................] ),
       ( Map [(”Part”, ”PC”),(”Year”, ”1997”)] ,
       Set[ Map [(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 4), (”Sale”, 6)],
            Map [(”City”, ”Mendoza”),(”Month”,”Feb”),(”Cost”, 4), (”Sale”, 8)],
            ....................................................................,
            Map [(”City”, ”Cordoba”),(”Month”, ”Jan”),(”Cost”, 4), (”Sale”, 8)],
            ....................................................................] ),
       ( Map [(”Part”, ”Inkjet”),(”Year”, ”1996”)] ,
       Set [ Map [(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 7), (”Sales”, 8)],
            Map [(”City”, ”Mendoza”),(”Month”, ”Feb”),(”Cost”, 7), (”Sale”, 9)],
            ....................................................................,
            Map [(”City”, ”Buenos Aires”),(”Month”, ”Jan”),(”Cost”, 6), (”Sale”, 9)],
            ....................................................................] ),
       ( Map [(”Part”, ”Inkjet”),(”Year”, ”1997”)] ,
       Set [ Map [(”City”, ”Mendoza”),(”Month”, ”Jan”),(”Cost”, 6), (”Sales”, 9)],
            Map [(”City”, ”Mendoza”),(”Month”, ”Feb”),(”Cost”, 6), (”Sale”, 8)],
            ....................................................................,
            Map [(”City”,”Buenos Aires”),(”Month”, ”Jan”),(”Cost”, 5), (”Sale”, 8)],
            ....................................................................] )
       ]
```

Table 4: Output of the expression $tcollect\{"Part","Year"\}(tuples\ RSALES)$.

5.2.2 Partial Function Application with a Default Value

Let apl be the isomorphism

$$A \rightharpoonup B \underset{apl^{-1}}{\overset{apl}{\cong}} (B+1)^A$$

in

$$\begin{aligned} get &: B \to A \to (A \rightharpoonup B) \to B \\ get\ u\ a\ f &\stackrel{\text{def}}{=} [id, \underline{u}] \cdot (apl\ f\ a) \end{aligned}$$

In Haskell:

```
get :: Eq a => b -> a -> Pfun a b ->  b
get u a f = aux (aplpf' f a)
            where aux (Ok b) = b
                  aux (Err s) = u
```

5.2.3 Tabular Reduction

Finally, function

$$\begin{aligned} ttot \quad &: \quad B \to ((A \times A) \to A) \to A \to \mathcal{P}(B \rightharpoonup A) \to (B \rightharpoonup A) \\ ttot\ b\ f\ u\ s \quad &\stackrel{\text{def}}{=} \quad \{b \mapsto tot2\ \ f\ \ u\ \ (\mathcal{P}(g\ \ u\ \ b))s\} \end{aligned}$$

performs tabular reduction, where $g\ \ u\ \ b \stackrel{\text{def}}{=} (id \times (get\ \ u\ \ b)) \cdot swap \cdot (tnest\ \ \{b\})$. Argument b specifies the measure attribute over which reduction will take place while arguments f and u provide the required reduction algebra. The output is packaged into a one-attribute tuple mapping the measure attribute name to the final result.

The corresponding Haskell code follows the above definition very closely:

```
ttot :: (Eq a, Eq b)
        => b
        -> (a -> a -> a)
        -> a
        -> Set (Pfun b a)
        -> Pfun b a
ttot b f u s = Map [ b |-> (tot2 f u (g u b s)) ]
     where g u b = nmap ((id >< (get u b)) . swap .
                         tnest (sings b))
```

Example 5.2 *Consider again the query of Example 4.1. We illustrate in this example how, from the classified set of tuples computed in Example 5.1, it is possible to obtain the final answer to the query.*

Let $fclass$ be the mapping arising from the classification step ($fclass$ type is $(Attr \rightharpoonup Value) \rightharpoonup \mathcal{P}(Attr \rightharpoonup Value)$) computed in Example 5.1. We can use $ttot$ to summarize over the Sales *attribute, with a particular binary operation (monoid $(+, 0)$ in this example), in the range of $fclass$. The last step is to transform the resulting structure in a table (set of tuples). Diagram (5) depicts the required computations.*

$$\begin{array}{c} \mathcal{P}(Attr \rightharpoonup Value) \\ \Big\downarrow{\scriptstyle tcollect\ \{Part,Year\}} \\ (Attr \rightharpoonup Value) \rightharpoonup \mathcal{P}(Attr \rightharpoonup Value) \\ \Big\downarrow{\scriptstyle id \rightharpoonup (ttot\ Sales\ (+)\ 0)} \\ (Attr \rightharpoonup Value) \rightharpoonup (Attr \rightharpoonup Value) \\ \Big\downarrow{\scriptstyle mkr} \\ \mathcal{P}((Attr \rightharpoonup Value) \times (Attr \rightharpoonup Value)) \\ \Big\downarrow{\scriptstyle \mathcal{P}(uncurry\ plus)} \\ \mathcal{P}(Attr \rightharpoonup Value) \end{array} \tag{5}$$

Altogether, we have evaluated the expression

$$(\mathcal{P}plus) \cdot mkr \cdot (id \rightharpoonup (ttot\ Sale\ (+)\ 0)) \cdot (tcollect\ \ \{Part, Year\}) \tag{6}$$

5.3 Multidimensional Analysis

Next we define a "multidimensional analysis" function that generalizes the algebraic structure of (6) above:

$$mda\ \ s\ \ a\ \ f\ \ u\ \stackrel{\text{def}}{=}\ (\mathcal{P}plus)\cdot mkr\cdot(B \rightharpoonup ttot\ \ a\ \ f\ \ u)\cdot(tcollect\ \ s)$$

In the context of our relational model in Haskell, we provide the mda function defined over the $Relation$ data type, as follows:

```
mdaR::      Set IdAttr
         -> IdAttr
         -> (Value -> Value -> Value)
         -> Value
         -> Relation
         -> Relation
mdaR s a f u  r =
         Rel ((unions s (sings a)) <:(schema r))
             (nmap (uncurry plus)(mkr y))
         where y = (id *-> (ttot a f u)) x
               x = tcollect s (tuples r)
```

Table 5 illustrates the application of the "multidimensional analysis" operation $mdaR$ to our running example. Operation $mdaR$ produces a relation with scheme $Part, Year, Sale$ which is depicted two-dimensionally.

PartYearSales		*Year*		
		1996	1997	...
Part	PC	320	455	...
	Inkjet	298	450	...
	⋮	⋮	⋮	

Table 5: Output of the expression `mdaR (Set ["Part","Year"] ) "Sale" fadd f0 RSALES` applied to the input relation *RSALES* of Example 4.1.

6 CONCLUDING REMARKS

The research carried out in this paper belongs to the intersection of formal methods with relational database theory.

In this paper we have presented a collection of functions in Haskell language that capture some of the functionality currently provided by multidimensional database products. We worked under the assumption that relational systems can model N-dimensional data as a relation with N-attribute domains.

Function mda, which creates a table with an aggregated value indexed by a set of attributes, operates on relations and produces relations. It could be composed with the basic operators of relational algebra to build other OLAP operators in order to provide constructs such histograms, cross-tabulations, subtotals, roll-up and drill-down. For instance, mda could be used to compute the following table (roll up using totals report):
The rightmost column corresponds to the output of the expression:

PartYearSales		*Year*			
		1996	1997	...	Total
Part	PC	320	455	...	1256
	Inkjet	298	450	...	987
	⋮	⋮	⋮	⋮	⋮
	Total	1788	1450	...	—

Table 6: Sales Roll Up by Part by Year (applied to the input relation *RSALES* of Example 4.1.

```
mdaR (Set["Part"]) "Sale" fadd f0 RSALES
```

while the bottom row could be computed using:

```
mdaR (Set["Year"]) "Sales" valadd val0 r1olap
```

Composition (and others categorical combinators) provides a powerful tool to compose operators and allows for complex multidimensional queries to be built.

It should be stressed that the operations defined do not intend to address the issue of restructuring information from the perspective of the dimensionality of the data.

The next step, which goes beyond the scope of the present work, should include the development of a complete algebra and a rigorous calculus based upon the algebraic operators of the model.

REFERENCES

[1] Rakesh Agrawal, A. Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In Alex Gray and Per-Åke Larson, editors, *Proc. 13th Int. Conf. Data Engineering, ICDE*, pages 232–243. IEEE Computer Society, 7–11 1997.

[2] E. F. Codd. *Missing Information*. Addison-Wesley Publishing Company, Inc., 1990.

[3] E. F. Codd. Providing olap (on-line analytical processing) to user-analyst: an it mandate, Apr. 1993. Technical Report, E. F. Codd and Associates.

[4] S. B. Codd E. F. Codd and C. T. Salley. Beyond decision support, July 1993. Computer World, 27:87–89.

[5] Marc Gyssens and Laks V. S. Lakshmanan. A foundation for multi-dimensional databases. In *The VLDB Journal*, pages 106–115, 1997.

[6] Chang Li and Xiaoyang Sean Wang. A data model for supporting on-line analytical processing. In *CIKM*, pages 81–88, 1996.

[7] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983. ISBN 0-914894-42-0.

[8] C. Necco. Polytypic data processing, may 2005. Master's thesis (Facultad de Cs. Físico Matemáticas y Naturales, University of San Luis, Argentina).

[9] J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspect of Computing*, 2(1):1–23, April 1990.

[10] J. N. Oliveira. A data structuring calculus and its application to program development, May 1998. Lecture Notes of M.Sc. Course Maestria em Ingeneria del Software, Departamento de Informatica, Facultad de Ciencias Fisico-Matematicas y Naturales, Universidad de San Luis, Argentina.

[11] Arie Shoshani. Statistical databases: Characteristics, problems, and some solutions. In *Eigth International Conference on Very Large Data Bases, September 8-10, 1982, Mexico City, Mexico, Proceedings*, pages 208–222. Morgan Kaufmann, 1982.