

Modeling, Querying and Reasoning about OLAP Databases: A Functional Approach

Ken Q. Pu
University of Toronto Toronto, ON, Canada
kenpu@cs.toronto.edu

ABSTRACT

We propose a new functional framework for modeling, querying and reasoning about OLAP databases. The framework represents data (data cubes and dimensional hierarchies) and querying constructs as first-order and second-order functional symbols respectively. A polymorphic attribute-based type system is used to annotate the functional symbols with proper type information. Furthermore, semantic knowledge about the functional symbols, such as the properties of dimensional hierarchical structures and algebraic identities among query constructs, can be specified by equations which permits equational reasoning on equivalence of OLAP queries and generalized summarizability of aggregate views.

Categories and Subject Descriptors

H.2 [Database Management]: Languages

General Terms

Languages, Theory

Keywords

OLAP, query-language, summarizability

1. INTRODUCTION

Polymorphic type system and higher-order functions have been proven to be highly effective in programming languages [25, 10]. These ideas have been applied to database management by several researchers [24, 4, 5]. We show how these ideas can be adopted with suitable generalization to model and query OLAP databases.

We use functions to represent data, queries and query constructs uniformly. Formally, functions are represented by *functional symbols*. The input and output of the functions are denoted by the type signature of the functional symbols, expressed in a polymorphic type system. Data is represented

as typed first-order functional symbols, each of which has an input (domain) and output (co-domain). Query constructs operate on data, and are modelled by second-order functional symbols. A second-order functional symbol accepts several functions as the input, and produces a function as its output.

Using the second-order functional symbols, one can form complex functional expressions which correspond to queries. *Type-checking* a query is to decide if all applications of the second-order functional symbols in the functional expression is sound with respect to the type system. In this way, an entire query language can be represented by a collection of typed functional symbols. One can show that existing OLAP query languages, such as these proposed in [1, 11], can all be modelled in this framework, and furthermore, new query constructs such as ordering and user-defined aggregates can be introduced as additional functional symbols. One major benefit of the functional representation of OLAP query languages is that we can perform static type analysis on the query expressions, which has not been dealt with before in the context of OLAP databases.

An equation is a pair of functional expressions. The functional expressions may have functional variables. An equation asserts that for any instantiation of the functional variables, the left hand-side is equivalent to the right hand-side. By the means of equations, we can specify and reason about equivalences of queries. As we later demonstrate, equational reasoning [19] offers a new solution to the problem of summarizability of aggregate views for a broader class of view definitions than the existing results on summarizability such as [16, 14].

We describe three advantages with our approach over existing OLAP formalisms.

- The query language, being a functional alphabet with equational specifications, is extensible. New functionalities, such as new aggregation functions, can be added into the query language as additional functional symbols and equations.
- All functions are polymorphically typed. Hence, we are able to statically perform type-checking and type-inference on OLAP queries. To our knowledge, static type analysis of OLAP queries has not been dealt with by existing formalisms.
- Equational specifications of the functional symbols permit us to deploy equational reasoning in the style of theorem-proving systems to solve some OLAP prob-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DOLAP'05, November 4–5, 2005, Bremen, Germany.
Copyright 2005 ACM 1-59593-162-7/05/0011 ...\$5.00.

lems such as summarizability. This nicely bridges syntactic language design and semantic query reasoning.

2. PRELIMINARIES: TYPES, FUNCTIONS AND EQUATIONS

2.1 The types

We use a polymorphic attribute-based type system to type functions. The type system is similar to type systems with records from the programming language community [8, 22, 27], as well as feature structures found in computational linguistics [9, 15]. A similar type system is applied for static type analysis of a specific database query language [5].

A record type expression, or simply a *type expression*, or more succinctly a *type*, consists of a collection of distinct attributes, each is further typed by either primitive types such as `string`, `numeric`, `bool` etc., or by another record type. For instance, we have the following record type expression of personal information:

$$t = \langle \text{Name} : \langle \text{Last} : \text{string}, \text{First} : \text{string} \rangle, \\ \text{Age} : \text{numeric}, \\ \text{Income} : \text{numeric} \rangle.$$

We can equivalently express it vertically as shown in Figure 1.

For the existing record-based type systems, *row variables* express extensibility of records, thus appear in the functional signature of polymorphic functions. Using row variables, one can express an extensible personal information type as t_1 in Figure 1, in which the row variable $\mathbf{x}^{\text{others}}$ can be instantiated to multiple fields arbitrarily.

Our experience shows, however, that these type systems still do not offer sufficient polymorphism to capture OLAP-related query constructs, so our type system generalizes the existing type systems by also allowing the use of *attribute variables*. An attribute variable can be used to specify an attribute whose name can be instantiated later. Consider type t_2 in Figure 1, it states that there are three attributes u_1, u_2 and u_3 whose names are to be specified, and u_1, u_2 are of the same type while u_3 is of type `string`. Types such as t_2 cannot be specified by row variables alone, but are needed when describing the type information of OLAP query constructs shown later in Figure 3.

Together, we refer to the row variables and the attribute variables as *type variables* to distinguish them from functional variables that appear in equational specifications that are defined later in Section 2.3.

Denote all type expressions by \mathbf{T} , the set of constant attribute names by \mathbf{A} , the set of attribute variables and row variables by \mathbf{V}^{attr} and \mathbf{V}^{row} respectively.

A type expression is *polymorphic* if it contains attribute or type variables, otherwise it is concrete. Let $t \in \mathbf{T}$ be polymorphic, namely with attribute or row variables. A *substitution* θ of t is a mapping on the variables that appear in t such that, for all attribute variables u in t , $\theta(u) \in \mathbf{V}^{\text{attr}} \cup \mathbf{A}$, and for all row variables \mathbf{x} in t , $\theta(\mathbf{x}) \in \mathbf{T}$. An *instance* of t is the type expression $\theta(t)$ derived from substituting types variables $x \in \mathbf{V}^{\text{attr}} \cup \mathbf{V}^{\text{row}}$ by $\theta(x)$ in t .

Let s and t be two polymorphic types, a unifier θ of the pair (s, t) is a substitution such that $\theta(s) = \theta(t)$. More generally, given a set of pairs $\{(s_i, t_i)\}$, θ is a unifier if $\theta(s_i) = \theta(t_i)$ for all i .

Unification has been well studied [23, 3]. One major application of unification is type-checking and type-inference [18] of programming languages. We also rely unification for type-checking and type-inference in our framework. The existing unification algorithms do not immediately apply to our type system since it allows polymorphic attribute names. Nevertheless, we are able to extend the equivalence-refinement approached in [20] to perform unification for our type system, which is used in static type analysis of OLAP queries. The detail and complexity of the unification algorithm is beyond the scope of this paper, and can be found in a separate report [21].

2.2 Functions and functional expressions

A *first-order* functional symbol f is of the form $f : s \rightarrow t$ where s and t are types in \mathbf{T} , which are referred to as the domain $\text{dom}(f)$ and the co-domain $\text{cod}(f)$ respectively. A *second-order* functional symbol α is of the form $\alpha : (s_1 \rightarrow t_1), (s_2 \rightarrow t_2), \dots (s_n \rightarrow t_n) \Rightarrow (s_0 \rightarrow t_0)$, where for all $i > 0$, s_i and t_i are types in \mathbf{T} . We refer to s_i and t_i as $\text{dom}_i(\alpha)$ and $\text{cod}_i(\alpha)$, respectively, and to n as the arity of α . A function is polymorphic if attribute or type variables are used in its signature. A polymorphic functional f can be instantiated by a substitution θ on type variables to a new functional symbol $f/\theta : \theta(\text{dom}(f)) \rightarrow \theta(\text{cod}(f))$. Instantiation of functions extends to second-order polymorphic functions naturally. Note that an instance of a polymorphic function is still polymorphic if there are type variables remain in its type signature, otherwise, we call it a concrete function.

EXAMPLE 1. A function that maps personal identification numbers (PIN) to the corresponding personal record is represented by the following first-order functional symbol and its type information.

$$\text{info} : \langle \text{PIN} : \text{string} \rangle \rightarrow \left[\begin{array}{l} \text{Name} : \left[\begin{array}{l} \text{Last} : \text{string} \\ \text{First} : \text{string} \end{array} \right] \\ \text{Age} : \text{numeric} \\ \text{Income} : \text{numeric} \end{array} \right.$$

The project function projects a record to some of its fields. It can be described as a first-order polymorphic function:

$$\text{project} : \langle \mathbf{x}^{\text{keep}}, \mathbf{x}^{\text{ignore}} \rangle \rightarrow \langle \mathbf{x}^{\text{keep}} \rangle$$

In order to make use of `project`, one must first instantiate the row variables \mathbf{x}^{keep} and $\mathbf{x}^{\text{ignore}}$ to indicate which fields to keep and which to be projected. This is true for polymorphic functions in general – their type variables must be instantiated before they can be used as part of a query. Often, it is not necessary to specify all the type variables, thanks to type-inference via unification. For the operator `project`, it's clear that one only needs to instantiate one of \mathbf{x}^{keep} and $\mathbf{x}^{\text{ignore}}$.

It's common to combine two functions by passing the output of one as the input of the other. This composition operation is a second-order function and is typed as follows:

$$\text{compose} : (\mathbf{x}^1 \rightarrow \mathbf{x}^2), (\mathbf{x}^2 \rightarrow \mathbf{x}^3) \Rightarrow (\mathbf{x}^1, \mathbf{x}^3)$$

Second-order functional symbols can be applied to first-order functional symbols to derive new functions. This is formally captured by functional expressions. A functional expression is either an instance of a first-order symbol, or an application of a second-order function $\alpha(e_1, e_2, \dots, e_n)$ where e_i are functional expressions, and α an instance of a

$$t = \left[\begin{array}{l} \text{Name} : \left[\begin{array}{l} \text{Last} : \text{string} \\ \text{First} : \text{string} \end{array} \right] \\ \text{Age} : \text{numeric} \\ \text{Income} : \text{numeric} \end{array} \right] \quad t_1 = \left[\begin{array}{l} \text{Name} : \left[\begin{array}{l} \text{Last} : \text{string} \\ \text{First} : \text{string} \end{array} \right] \\ \text{Age} : \text{numeric} \\ \text{Income} : \text{numeric} \\ \mathbf{x}^{\text{others}} \end{array} \right] \quad t_2 = \left[\begin{array}{l} \text{Name} : \left[\begin{array}{l} \text{Last} : \text{string} \\ \text{First} : \text{string} \end{array} \right] \\ \text{Age} : \text{numeric} \\ \text{Income} : \text{numeric} \\ u^1 : \mathbf{x}^1 \\ u^2 : \mathbf{x}^1 \\ u^3 : \text{string} \\ \mathbf{x}^{\text{others}} \end{array} \right]$$

Figure 1: Examples of record types

second-order functional symbol with arity n . Let Σ be an alphabet of functional symbols, then we write $\mathcal{L}(\Sigma)$ as the functional expressions built using symbols from Σ . We can speak of polymorphic and concrete functional expressions – an expression is polymorphic if it makes use of one or more polymorphic functions, otherwise it is concrete.

Since functions are typed, not all expression in $\mathcal{L}(\Sigma)$ are valid according to the type information. Consider the expression

compose(info, info)

which composes the output of `info` with the input of `info`. Clearly, it is ill-typed expression. The validity of a functional expression is captured by the notions of *well-typedness* and *valid-typedness*. An expression is *well-typed* if it is a first-order functional symbol, or it is of the form $\alpha(e_1, e_2, \dots, e_n)$ where $\alpha : (s_1 \rightarrow t_1), (s_2 \rightarrow t_2), \dots, (s_n \rightarrow t_n) \Rightarrow (s_0 \rightarrow t_0)$, and for all i , the expression e_i is well-typed, and of the form $e_i : s_i \rightarrow t_i$. The following is a well-typed expression:

Let θ_1 be:

$$\theta_1(\mathbf{x}^1) = \langle \text{PIN} : \text{string} \rangle, \quad \theta_1(\mathbf{x}^2) = \left[\begin{array}{l} \text{Name} : \left[\begin{array}{l} \text{Last} : \text{string} \\ \text{First} : \text{string} \end{array} \right] \\ \text{Age} : \text{numeric} \\ \text{Income} : \text{numeric} \end{array} \right]$$

and θ_2 be:

$$\theta_2(\mathbf{x}^{\text{keep}}) = \langle \text{Income} : \text{numeric} \rangle, \quad \theta_2(\mathbf{x}^{\text{ignore}}) = \left[\begin{array}{l} \text{Name} : \left[\begin{array}{l} \text{Last} : \text{string} \\ \text{First} : \text{string} \end{array} \right] \\ \text{Age} : \text{numeric} \end{array} \right]$$

Then, `compose/ θ_1 (info, project/ θ_2)` is a well-typed expression.

Well-typedness requires many needless instantiations of type variables since it requires the signatures to match exactly. An expression e is *validly-typed* if there exists an instantiation on the type variables used e such that e becomes well-typed. Therefore, valid-typed expressions do not require exact matching of types. The main difference between well-typedness and validly-typedness is that a well-typed query is guaranteed to be type correct, while a valid-typed query does not contain a type-error, but future instantiations of polymorphic type variables may still potentially introduce type-errors. In other words, an expression is validly-typed if no type-error has been committed. Therefore,

$$\text{compose}(\text{info}, \text{project} / \{ \mathbf{x}^{\text{keep}} = \langle \text{Income} : \text{numeric} \rangle \})$$

is validly-typed, but not well-typed. Deciding whether an expression is validly typed amounts to unification of the type signature of the arguments with the type signature of the operator.

2.3 Equations

In this section, we informally introduce equations. Standard treatment of equations and equational equivalence are

found in [2, 12]. We require the slight generalization of associating types to the functional symbols and variables.

An equation E is of the form $E = (\lambda f_1 : s_1 \rightarrow t_1, f_2 : s_2 \rightarrow t_2, \dots)$. $e_1 \simeq e_2$, where $f_i : s_i \rightarrow t_i$ are functional variables disjoint from the functional alphabet, and e_1 and e_2 are two validly-typed functional expressions that make use of the variables f_i . It is required that the two functions e_1 and e_2 have the same type signature.

Equations can be thought of as query-rewrite rules that transforms a query into an equivalent form. Given a query q as a functional expression, and equation $E = (\lambda \dots)$. $e_1 \simeq e_2$, E can be applied to q if there exists a substitution θ of variables in e_1 (respectively e_2) such that $\theta(e_1)$ (resp. $\theta(e_2)$) is a sub-query in q . In this case, we can replace the sub-query $\theta(e_1)$ in q with $\theta(e_2)$ to derive an equivalent query q' . And we write $q \leftrightarrow_E q'$.

An *equational theory* \mathcal{E} is a set of equations. The equational equivalence $\equiv_{\mathcal{E}}$ defined by \mathcal{E} is equivalence on queries, given as $q \equiv_{\mathcal{E}} q'$ if q can be transformed into q' using the equations in \mathcal{E} .

3. MODELING AND QUERYING OLAP DATABASES

The idea of treating OLAP cubes as functions mapping from a cartesian product space of dimensional attributes to a space of measure values has been proposed by several researchers [1, 6, 7]. In our framework, an OLAP cube is naturally represented as a concrete functional symbol with the dimensions and the names of the measures as attributes of the domain and co-domain respectively. We can also represent the outline of the dimensional hierarchies using functions with type signatures using collection types.

Modeling a data cube: An OLAP cube with dimensions d_1, d_2, \dots, d_n , and measures m_1, m_2, \dots, m_k is a functional symbol of the form

$$\text{cube} : \langle d_1 : s_1, d_2 : s_2, \dots, d_n : s_n \rangle \rightarrow \langle m_1 : t_1, m_2 : t_2, \dots, m_k : t_k \rangle$$

where s_i and t_j are primitive types of the dimension d_i and measure m_j respectively.

Modeling a hierarchical dimension: A classification function that groups category c_j to a category of c_i is a functional symbol of the form

$$\downarrow_{c_j}^{c_i} : \langle c_i : s_i \rangle \rightarrow \langle c_j : \text{coll}(c_j : t_j) \rangle$$

where `coll` can be any of `set`, `bag` or `list`.¹ A hierarchical

¹A collection `set(t)` presents a set of objects, each of type t . Similarly, `bag(t)` and `list(t)` are bags and lists of objects of type t . The semantics is the usual one. Lists are ordered and with duplicates, bags are unordered, and sets are unordered and with no duplicates.

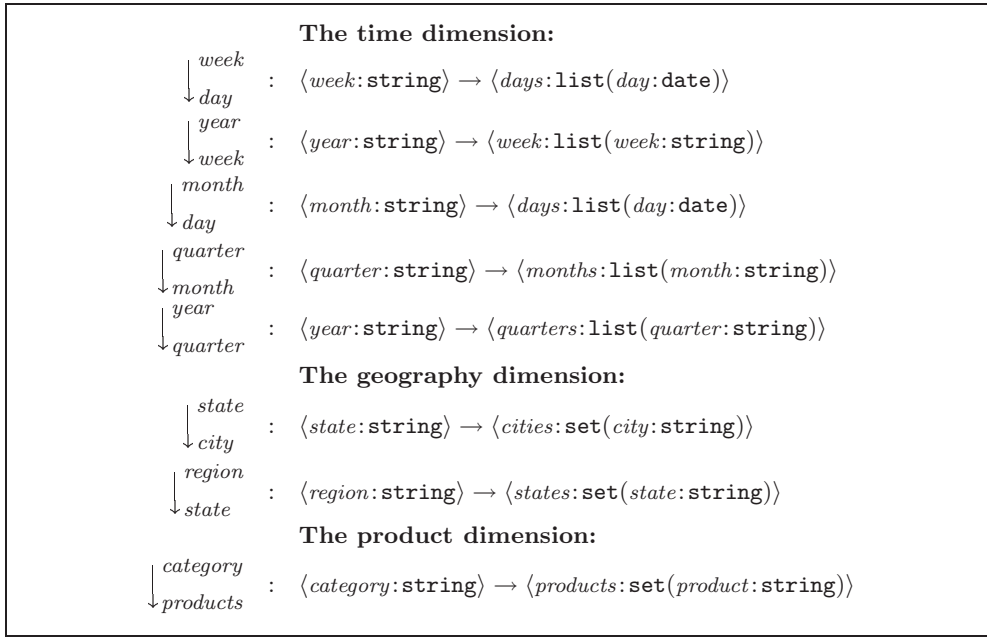


Figure 2: Functional symbols representing dimensional outline.

dimension D is simply a collection classification functions $\{\downarrow_{c_i}\}$ where c_i, c_j are categories in the dimension D . Often, we have a distinguished *all-category* D^{top} , and a function $\text{all}_D : \text{unit} \rightarrow D^{\text{top}}$.

Modeling constants: Sometimes, in addition to data cubes and classification functions, one also require constants to query OLAP databases. For instance, a string constant such as ‘‘Chicago’’ needs to be explicitly mentioned in a query. A string constant is a function of the form $c : \text{unit} \rightarrow u : \text{string}$. Notice that a string constant is in fact a polymorphic function – this is because our type system is attribute-based, so one needs to specify the attribute of the string. For instance, ‘‘Chicago’’ can be used as a city name as ‘‘Chicago’’/ $\{u = \text{city}\} : \text{unit} \rightarrow \langle \text{city}:\text{string} \rangle$, or a movie name as in ‘‘Chicago’’/ $\{u = \text{movie}\} : \text{unit} \rightarrow \langle \text{movie}:\text{string} \rangle$.

EXAMPLE 2. Consider an OLAP database that keeps track of performances of sales figures of stores across United States for various products over time.

There are two data cubes. The first stores the sales figures.

$$\text{sales} : \begin{bmatrix} \text{day}:\text{date}, \\ \text{city}:\text{string}, \\ \text{product}:\text{string} \end{bmatrix} \rightarrow \begin{bmatrix} \text{unit sold}:\text{numeric}, \\ \text{inventory}:\text{numeric}, \\ \text{expense}:\text{numeric} \end{bmatrix}$$

The three dimensions for sales are day, city and product. The measures are unit sold, inventory and expense which correspond to the units of products sold, the current inventory, and the operating expense of that day.

The second data cube stores the pricing of the products. The price of a product is set daily, and is independent of the geography.

$$\text{price} : \langle \text{day}:\text{string}, \text{product}:\text{string} \rangle \rightarrow \langle \text{price}:\text{numeric} \rangle$$

In the time dimension, days are grouped into weeks and months. weeks are grouped into years, while months are grouped

into quarters and then years. In the geography dimension, cities are grouped into states and then regions. Finally, in the product dimension, we group the products into categories. The classifications are captured by the functions shown in Figure 2.

Note that for the time dimension, we use `list` collection types, whereas for the geography and product dimensions, the `set` collection is used. This is for the simple reason that there is a linear order to categories (day, week, etc.) in the time dimension.

These functional symbols encode the outlines of the dimensions, but do not reflex the structural constraints such as the notion of strict and non-strict hierarchies [13]. For instance, intuitively, we understand that $\downarrow_{\text{state}}^{\text{region}}$ should form a partition (a strict hierarchy), that is, two regions cannot contain the same state; same applies to the classification functions in the time dimension such as $\downarrow_{\text{day}}^{\text{week}}$. However,

for the classification of $\downarrow_{\text{products}}^{\text{category}}$, one can imagine that the same product belongs to two different categories, hence it is a non-strict hierarchy.

The partitioning constraints of the classification functions can be expressed by equations which is the subject of equational specification in Section 4.

3.1 OLAP query language as a functional alphabet

In this section, we show how polymorphic functions are used to model query languages. In particular, we present a collection \mathbf{Q}^{olap} of query constructs in forms of polymorphic first- and second-order functions. Queries are then functional expressions in $\mathcal{L}(\mathbf{Q}^{\text{olap}} \cup \Sigma)$ formed by query constructs from \mathbf{Q}^{olap} and data functions from the schema Σ .

EXAMPLE 3. Consider the sales data cube in Example 2 and suppose we want to obtain from sales a new data cube

OLAP query constructs Q^{olap} :	
construct	$\langle \mathbf{x} \rangle \rightarrow \langle u : \mathbf{x} \rangle$ destruct : $\langle u : \mathbf{x} \rangle \rightarrow \langle \mathbf{x} \rangle$
project	$\langle \mathbf{x}^{\text{keep}}, \mathbf{x}^{\text{ignore}} \rangle \rightarrow \langle \mathbf{x}^{\text{keep}} \rangle$
sng ^{coll}	$\langle \mathbf{x} \rangle \rightarrow u : \text{coll}(\mathbf{x})$
\cup^{coll}	$\langle u_1 : \text{coll}(\mathbf{x}), u_2 : \text{coll}(\mathbf{x}) \rangle \rightarrow \langle u_3 : \text{coll}(\mathbf{x}) \rangle$
\cap^{coll}	$\langle u_1 : \text{coll}(\mathbf{x}), u_2 : \text{coll}(\mathbf{x}) \rangle \rightarrow \langle u_3 : \text{coll}(\mathbf{x}) \rangle$
\setminus^{coll}	$\langle u_1 : \text{coll}(\mathbf{x}), u_2 : \text{coll}(\mathbf{x}) \rangle \rightarrow \langle u_3 : \text{coll}(\mathbf{x}) \rangle$
set_to_bag	$\langle u : \text{set}(\mathbf{x}) \rangle \rightarrow \langle u : \text{bag}(\mathbf{x}) \rangle$
list_to_bag	$\langle u : \text{list}(\mathbf{x}) \rangle \rightarrow \langle u : \text{bag}(\mathbf{x}) \rangle$
flatten ^{coll}	$\langle u_1 : \text{coll}(u_2 : \text{coll}(\mathbf{x})) \rangle \rightarrow \langle u^{\text{out}} : \text{coll}(\mathbf{x}) \rangle$
compose	$\langle \mathbf{x}^1 \rangle \rightarrow \langle \mathbf{x}^{\text{compose}}, \mathbf{x}^{\text{ignore}} \rangle, \langle \mathbf{x}^{\text{compose}}, \mathbf{x}^2 \rangle \rightarrow \langle \mathbf{x}^{\text{final}} \rangle \Rightarrow \langle \mathbf{x}^1, \mathbf{x}^2 \rangle \rightarrow \langle \mathbf{x}^{\text{final}} \rangle$
product	$\langle \mathbf{x}^{\text{common}}, \mathbf{x}^1 \rangle \rightarrow \langle \mathbf{x}^{\text{left}} \rangle, \langle \mathbf{x}^{\text{common}}, \mathbf{x}^2 \rangle \rightarrow \langle \mathbf{x}^{\text{right}} \rangle$ $\Rightarrow \langle \mathbf{x}^{\text{common}}, \mathbf{x}^1, \mathbf{x}^2 \rangle \rightarrow \langle \mathbf{x}^{\text{left}}, \mathbf{x}^{\text{right}} \rangle$
map ^{coll}	$\langle \mathbf{x}^{\text{loop}}, \mathbf{x}^{\text{aux}} \rangle \rightarrow \langle \mathbf{x}^{\text{final}} \rangle \Rightarrow \langle u^{\text{loop}} : \text{coll}(\mathbf{x}^{\text{loop}}), \mathbf{x}^{\text{aux}} \rangle \rightarrow \langle u^{\text{final}} : \text{coll}(\mathbf{x}^{\text{final}}) \rangle$
filter ^{coll}	$\langle \mathbf{x}^{\text{loop}}, \mathbf{x}^{\text{aux}} \rangle \rightarrow \text{bool} \Rightarrow \langle u_1 : \text{coll}(\mathbf{x}^{\text{loop}}), \mathbf{x}^{\text{aux}} \rangle \rightarrow \langle u_2 : \text{coll}(\mathbf{x}^{\text{loop}}) \rangle$
sum	$\langle u_1 : \text{bag}(u_2 : \text{numeric}) \rangle \rightarrow \langle u^{\text{out}} : \text{numeric} \rangle$
min	$\langle u_1 : \text{set}(u_2 : \text{numeric}) \rangle \rightarrow \langle u^{\text{out}} : \text{numeric} \rangle$
first	$\langle u_1 : \text{list}(u_2 : \text{numeric}) \rangle \rightarrow \langle u^{\text{out}} : \text{numeric} \rangle$
+ / - *	$\langle u_1 : \text{numeric}, u_2 : \text{numeric} \rangle \rightarrow \langle u^{\text{out}} : \text{numeric} \rangle$

Figure 3: OLAP Query Constructs

expense that only has one measure expense. The cube expense can be derived by composing (using the composition operator `compose`) with the `project` operator `project`. The type signatures of the `compose` and `project` operators are shown in Example 1. One is tempted to define `expense` as the query q_1 :

$$q_1 = \text{compose}(\text{sales}, \text{project}). \quad (7)$$

However, we have not identified which measure(s) to keep, so q_1 is ambiguously defined. The ambiguity can be resolved by not using `project` but its instance `project/{xkeep = expense:numeric}`. Thus, we now define `expense` as the query q_2 :

$$q_2 = \text{compose}(\text{sales}, \text{project}/\{\mathbf{x}^{\text{keep}} = \text{expense:numeric}\}). \quad (8)$$

Query q_2 is sufficiently specified despite the fact that type variables such as $\mathbf{x}^{\text{ignore}}$ in the signature of `project` and $\mathbf{x}^1, \mathbf{x}^2$ and \mathbf{x}^3 in the signature of `compose` are left unspecified. The instantiation of the unspecified type variables can be uniquely inferred from the fact that q_2 must be validly typed.

Figure 3 lists the set of OLAP query constructs. The semantics is clear for most of the functions. We only briefly describe informally the semantics of the more complex query constructs. The `compose` operator generalizes the functional composition by allowing mismatches between the output and the input of the two composed functions. The `mapcoll` operator is such that given an input function f , the output function `mapcoll(f)` applies f to each of the elements in the

collection associated with the attribute u^{loop} , and produces a new collection $\langle u^{\text{final}} : \mathbf{x}^{\text{final}} \rangle$. The `filtercoll` operator takes a predicate function p as an input, and `filtercoll(p)` maps a collection of elements to those that the predicate p is true. It's not too difficult to see that the query language $\mathcal{L}(Q^{\text{olap}})$ is relational-complete.

It can be quite cumbersome to express queries as functional expressions, so for presentation brevity (as well as usability), we define a set of syntactic short-hands.

- **Composition:** Often one wants to compose several functions, say f_1, f_2 and f_n . Instead of

$$\text{compose}(f_1, \text{compose}(f_2, f_3)),$$

we write using infix notion as $f_3 \circ (f_2 \circ f_1)$. In fact, the parenthesis are always assumed to be right-most, so one can simply write it as $f_3 \circ f_2 \circ f_1$.

- **Product:** Taking the product of several functions is also a common task. Rather than writing

$$\text{product}(f_1, \text{product}(f_2, f_3)),$$

we write it as $f_1 \times f_2 \times f_3$.

We illustrate how these operators are used to query an OLAP database by some examples.

Some equational specifications for OLAP query language:	
$(\lambda f_1 : \langle \mathbf{x}^{in1} \rangle \rightarrow \langle \mathbf{x}^{out1} \rangle, f_2 : \langle \mathbf{x}^{out1}, \mathbf{x}^{in2} \rangle \rightarrow \langle \mathbf{x}^{out2} \rangle,$	(1)
$f_3 : \langle \mathbf{x}^{out2}, \mathbf{x}^{in3} \rangle \rightarrow \langle \mathbf{x}^{out3} \rangle). f_3 \circ (f_2 \circ f_1) \simeq (f_3 \circ f_2) \circ f_1$	
$(\lambda f_1 : \langle \mathbf{x}^1, \mathbf{x}^{aux1} \rangle \rightarrow \langle \mathbf{x}^2 \rangle, f_2 : \langle \mathbf{x}^2, \mathbf{x}^{aux2} \rangle \rightarrow \langle \mathbf{x}^3 \rangle).$	(2)
$\text{map}/\{\mathbf{x}^{loop} = \mathbf{x}^1\}(f_2 \circ f_1) \simeq \text{map}/\{\mathbf{x}^{loop} = \mathbf{x}^2\}(f_2) \circ \text{map}/\{\mathbf{x}^{loop} = \mathbf{x}^1\}(f_1)$	
$(\lambda f : \langle \mathbf{x}^1 \rangle \rightarrow \langle \mathbf{x}^2 \rangle). \text{map}(f) \circ \text{flatten} \simeq \text{flatten} \circ \text{map}(\text{map}(f))$	(3)
$\text{sum} \circ \text{flatten}^{\text{bag}} = \text{sum} \circ \text{map}^{\text{bag}}(\text{sum})$	(4)
$\text{min} \circ \text{flatten}^{\text{set}} = \text{min} \circ \text{map}^{\text{set}}(\text{min})$	(5)
Equational specifications for OLAP dimensional structure:	
$\text{set_to_bag} \circ \text{flatten}^{\text{set}} \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right)$	(6)
$\simeq \text{flatten}^{\text{bag}} \circ \text{map}^{\text{bag}}(\text{set_to_bag} \circ \downarrow_{\text{city}}^{\text{state}}) \circ \text{set_to_bag}$	

Figure 4: Some equational specifications on OLAP query constructs

A simple select query: *Retrieve the units sold of Pepsi in Chicago on the day of Jan/01/2000.*

$$\begin{aligned}
q &= \text{project}/\{\mathbf{x}^{\text{keep}} = \langle \text{units sold:numeric} \rangle\} \circ \text{sales} \circ \\
&\quad (\text{"Pepsi"}/\{u = \text{product}\} \\
&\quad \times \text{"Chicago"}/\{u = \text{city}\} \\
&\quad \times \text{"Jan/01/2000"}/\{u = \text{date}\})
\end{aligned}$$

Note that q as defined above is a functional expression with the type signature of $\text{unit} \rightarrow \langle \text{units sold:numeric} \rangle$.

An aggregation query: *Derive the aggregate view of units sold at the state and year level.*

First, we would like define a query that maps a year to a list of days.

$$\downarrow_{\text{days}}^{\text{year}} = \text{flatten}^{\text{list}}/\{u^{\text{out}} = \text{days}\} \circ \text{map}^{\text{list}} \left(\downarrow_{\text{days}}^{\text{month}} \right) \circ \downarrow_{\text{months}}^{\text{year}}$$

One can verify that the type signature of $\downarrow_{\text{days}}^{\text{year}}$ is

$$\langle \text{year: string} \rangle \rightarrow \langle \text{days: list}(\text{day: date}) \rangle.$$

Next we define a function that simultaneously maps a year to a bag of days and a state to a bag of cities:

$$\Downarrow = \left(\text{list_to_bag} \circ \downarrow_{\text{days}}^{\text{year}} \right) \times \left(\text{set_to_bag} \circ \downarrow_{\text{cities}}^{\text{state}} \right)$$

Then we apply the cube function sales to the output of \Downarrow :

$$\begin{aligned}
q &= \text{map}^{\text{bag}}\{\mathbf{x}^{\text{loop}} = \langle \text{day: date} \rangle\} \\
&\quad \left(\text{map}^{\text{bag}}\{\mathbf{x}^{\text{loop}} = \langle \text{city: string} \rangle\}(\text{sales}) \right) \circ \Downarrow
\end{aligned}$$

The signature of q is

$$\begin{aligned}
q &: \langle \text{year: string}, \text{state: string}, \text{product: string} \rangle \\
&\rightarrow \langle u_1 : \text{bag}(u_2 : \text{bag}(\text{units sold : numeric})) \rangle.
\end{aligned}$$

Note that q has two attribute variables in its co-domain. They can be eliminated by specifying the u^{final} for the two map -operators, but in this case, it doesn't really matter since we can continue with the aggregation just fine. The aggregation view is then:

$$\text{sum} \circ \text{flatten}^{\text{bag}} \circ q.$$

Derived measure: *Derive the revenue measure. The revenue is defined to be the product of unit sold measure in the sales cube and the price measure in the*

price cube.

$$\begin{aligned}
q &= */\{u^{\text{out}} = \text{rev}\} \circ \\
&\quad ((\text{project}/\{\mathbf{x}^{\text{keep}} = \langle \text{units sold:numeric} \rangle\} \circ \text{sales}) \times \text{project})
\end{aligned}$$

The type signature of q is

$$q : \langle \text{day: string}, \text{city: string}, \text{product: string} \rangle \rightarrow \langle \text{rev: numeric} \rangle$$

4. EQUATIONAL SPECIFICATIONS OF OLAP QUERIES

In this section, we show how equations can be used to axiomatize OLAP functional symbols. A collection of equational specifications are given in Figure 4. Equation 1 – Equation 5 deal with the algebraic properties of the query constructs, while Equation 6 describes the structure of the dimensional hierarchy for the *region* dimension of the OLAP database.

Let's explain some of the equations. Equation 1 is simple; it states that the composition \circ is associative. Equation 2 states that map can be distributed over composition (\circ). These equations are similar to monad laws in functional programming [26], but one should note, however, that Equation 2 does not hold if the map loops over the type $\mathbf{x}^{\text{aux}2}$ in the signature of f_2 . So, the type constraints on the functional variables make these equations somewhat different from monad laws.

Equation 6 looks rather complex, but one can verify that it states exactly the fact that $\downarrow_{\text{city}}^{\text{state}}$ is a partition, i.e., every city belongs to only one state. Similarly,

The equational specifications on the functional symbols allow us to reason about equivalent forms of a given query. This offers a solution to the classical problem of answering query with views [17], a special case of which is the OLAP summarizability problem [14].

4.1 Applying equations to solve summarizability

We view summarizability as a special case of answering queries with views. We suppose a set of views $V = \{v_1, v_2, \dots\}$ defined as functional expressions. Treat v_i as functional symbols, their definitions $v_i = \dots$ are then equations. Now,

$$\begin{aligned}
& \text{sum} \circ \text{map}^{\text{bag}}(\text{cube}) \circ \text{flatten}^{\text{bag}} \circ \text{set_to_bag} \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right) \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(6)} \rightsquigarrow & \text{sum} \circ \underline{\text{map}^{\text{bag}}(\text{cube}) \circ \text{flatten}^{\text{bag}}} \circ \text{map}^{\text{bag}}(\text{set_to_bag} \circ \downarrow_{\text{city}}^{\text{state}}) \circ \text{set_to_bag} \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(5)} \rightsquigarrow & \underline{\text{sum} \circ \text{flatten}^{\text{bag}}} \circ \text{map}^{\text{bag}}(\text{map}^{\text{bag}}(\text{cube})) \circ \text{map}^{\text{bag}}(\text{set_to_bag} \circ \downarrow_{\text{city}}^{\text{state}}) \circ \text{set_to_bag} \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(6)} \rightsquigarrow & \text{sum} \circ \text{map}^{\text{bag}}(\text{sum}) \circ \text{map}^{\text{bag}}(\text{map}^{\text{bag}}(\text{cube})) \circ \text{map}^{\text{bag}}(\text{set_to_bag} \circ \downarrow_{\text{city}}^{\text{state}}) \circ \text{set_to_bag} \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(4)} \rightsquigarrow & \text{sum} \circ \text{map}^{\text{bag}}(\text{sum} \circ \text{map}^{\text{bag}}(\text{cube}) \circ \text{set_to_bag} \circ \downarrow_{\text{city}}^{\text{state}}) \circ \text{set_to_bag} \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(9)} \rightsquigarrow & \text{sum} \circ \text{map}^{\text{bag}}(v) \circ \text{set_to_bag} \circ \downarrow_{\text{state}}^{\text{region}}.
\end{aligned}$$

Figure 5: Incremental rewrite of query by applications of equations.

given a user defined query, q_0 , which uses the row-data cubes, we ask if q_0 is equivalent to some other query q' such that q' only makes use of query constructs in \mathbf{Q}^{olap} and the view symbols V , namely $q' \in \mathcal{L}(\mathbf{Q}^{\text{olap}} \cup V)$.

In search of a rewrite q' of the query q_0 , we iteratively apply the equational specifications of the functional symbols to explore equivalent forms of q_0 until we either found an equivalent form in $\mathcal{L}(\mathbf{Q}^{\text{olap}} \cup V)$ or exhausted all possible forms. This is similar to search techniques in automated reasoning tools such as Otter [19]. Of course, this is only a semi-decidable procedure in general, but for the given equational specifications shown in Figure 4, every query has only finitely many equivalent forms, thus the search procedure is guaranteed to terminate.

The following example illustrates a search sequence.

EXAMPLE 4. Consider an OLAP cube represented by the functional symbol `cube` as follows.

$$\text{cube} = \text{project}/\{\mathbf{x}^{\text{keep}} = \langle \text{units sold} : \text{numeric} \rangle\} \circ \text{sales}.$$

The view is the aggregate view at the state level:

$$v = \text{sum} \circ \text{map}^{\text{bag}}(\text{cube}) \circ \text{set_to_bag} \circ \downarrow_{\text{city}}^{\text{state}}. \quad (9)$$

Note that since the aggregation function is `sum`, which can only act on bags, we must use `set_to_bag` to convert the set from the output of $\downarrow_{\text{city}}^{\text{state}}$ to a bag. The user-defined query is to aggregate to the region level directly from the city level:

$$q_0 = \text{sum} \circ \text{map}^{\text{bag}}(\text{cube}) \circ \text{flatten}^{\text{bag}} \circ \text{set_to_bag} \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right) \circ \downarrow_{\text{state}}^{\text{region}}.$$

We apply the equations to transform q_0 into its equivalent forms. The steps are shown in Figure 5.

At each step, the rewritten sub-query is underlined. We omitted applications of the associativity of composition (\circ).

In the previous example, the partition property of $\downarrow_{\text{city}}^{\text{state}}$, Equation 6, is necessary in the sequence of rewritings. We now consider a similar query with the aggregation function `min`.

EXAMPLE 5. Again, we consider `cube = project/{ $\mathbf{x}^{\text{keep}} = \langle \text{units sold} : \text{numeric} \rangle$ } \circ \text{sales}`. The view is the aggregate view

at the state level, with the aggregation being `min`. Note that `min` can be applied over sets, thus, unlike as it was the case in Equation 9, we no longer need to convert set to bag `set_to_bag`.

$$v = \text{min} \circ \text{map}^{\text{set}}(\text{cube}) \circ \downarrow_{\text{city}}^{\text{state}}. \quad (10)$$

The user-defined query is to aggregate to the region level directly from the city level:

$$q_0 = \text{min} \circ \text{map}^{\text{set}}(\text{cube}) \circ \text{flatten}^{\text{set}} \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right) \circ \downarrow_{\text{state}}^{\text{region}}.$$

We apply the equations to transform q_0 :

$$\begin{aligned}
& \text{min} \circ \underline{\text{map}^{\text{set}}(\text{cube}) \circ \text{flatten}^{\text{set}}} \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right) \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(5)} \rightsquigarrow & \underline{\text{min} \circ \text{flatten}^{\text{set}}} \circ \text{map}^{\text{set}}(\text{map}^{\text{set}}(\text{cube})) \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right) \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(7)} \rightsquigarrow & \text{min} \circ \text{map}^{\text{set}}(\text{min}) \circ \text{map}^{\text{set}}(\text{map}^{\text{set}}(\text{cube})) \circ \text{map}^{\text{set}} \left(\downarrow_{\text{city}}^{\text{state}} \right) \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(4)} \rightsquigarrow & \text{min} \cdot \text{map}^{\text{set}}(\text{min} \circ \text{map}^{\text{set}}(\text{cube}) \circ \downarrow_{\text{city}}^{\text{state}}) \circ \downarrow_{\text{state}}^{\text{region}} \\
\text{Eq.(10)} \rightsquigarrow & \text{min} \cdot \text{map}^{\text{set}}(v) \circ \downarrow_{\text{state}}^{\text{region}}
\end{aligned}$$

Therefore q_0 is summarizable from v , as expected.

Note the summarizability in Example 5 cannot be obtained by approach in [14] which only considers aggregation by summation. We can treat the aggregation `min` because it is equationally specified. This means that other aggregation functions can be dealt with similarly.

5. CONCLUSION AND FUTURE WORK

We presented a framework in which functional symbols annotated by typing information are used to represent OLAP databases and their query language. Query constructs are higher-order polymorphic functions, and queries are expressed as complex functional expression. We demonstrated that the query language is flexible enough to express useful OLAP queries, and can be axiomatized by equational specifications. The equational specifications can aid us to automatically reason about problems such as summarizability of OLAP views.

We only suggested that the equational derivation with respect to the equational theory shown in Figure 4 terminates.

As on-going work, we are studying the more precise computational complexity of the decision procedure. Furthermore, we would like to characterize a class of queries for which one can decide summarizability in PTIME.

Our future work also includes investigating a broader family of equational specifications and the complexity of the equational reasoning over OLAP-specific equational axioms.

6. REFERENCES

- [1] R. Agarwal, A. Gupta, and S. Sarawagi. Modeling multidimensional databases. In *ICDE*, pages 232–243, 1997.
- [2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [3] Franz Baader and Wayne Snyder. Unification theory. *Handbook of automated reasoning*, pages 445–532, 2001.
- [4] Peter Buneman and Robert E. Frankel. FQL – a functional query language. In *SIGMOD*, pages 52–58, 1979.
- [5] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.
- [6] L. Cabbibo and R. Torlone. Querying multidimensional databases. In *6th DBPL Workshops*, 1997.
- [7] L. Cabbibo and R. Torlone. A logical approach to multidimensional databases. In *Proc. of the EDBT*, 1998.
- [8] Luca Cardelli and John C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991.
- [9] Bob Carpenter. *The Logic of Typed Feature Structures*. Cambridge University Press, 1992.
- [10] Paul Graham. *ANSI Common LISP*. Prentice-Hall, 1995.
- [11] M. Gyssens and L.V.S. Lakshmanan. A foundation for multidimensional databases. In *Proc. of the VLDB*, 1997.
- [12] Steffen Hölldobler. *Foundations of Equational Logic Programming*. Springer-Verlag New York, Inc., 1989.
- [13] John Horner, Il-Yeol Song, and Peter P. Chen. An analysis of additivity in olap systems. In *DOLAP'04*, pages 83 – 92, 2004.
- [14] C.A. Hurtado and A.O. Mendelzon. Reasoning about summarizability in heterogeneous multidimensional schemas. In *ICDT*, 2001.
- [15] Robert Kasper and William C. Rounds. A logical semantics for feature structures. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 257–266. Association for Computational Linguistics, 1986.
- [16] Hans-Joachim Lenz and Arie Shoshani. Summarizability in olap and statistical data bases. In *SSDBM '97: Proceedings of the Ninth International Conference on Scientific and Statistical Database Management*, pages 132–143, 1997.
- [17] A.Y. Levy, A.O. Mendelzon, and Y. Sagiv. Answering queries using views. In *PODS*, 1995.
- [18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17, 1978.
- [19] Otter. <http://www-unix.mcs.anl.gov/AR/otter/>.
- [20] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16, 1978.
- [21] Ken Q. Pu. Complexity and algorithm of unification of attribute-based type-expressions. Technical report, Department of Computer Science, University of Toronto, 2005. http://www.cs.toronto.edu/~kenpu/TR_unify.pdf.
- [22] Didier Rémy. Type inference for records in a natural extension of ML. In *TACS*, volume 789 of *Lecture Notes in Computer Science*. Springer, 1994.
- [23] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41, 1967.
- [24] David W. Shipman. The functional data model and the data languages DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.
- [25] Jeffery Ullman. *Elements of ML Programming*. Prentice-Hall, 1997.
- [26] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, 1995.
- [27] M. Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, 1987.