HIT DATA MODEL
DATA BASES FROM THE FUNCTIONAL POINT OF VIEW

Jiří Zlatuška

Comp. Sci. Dept. University of Brno
Kotlářská 2, CS-611 37 Brno, Czechoslovakia

Abstract
Basic notions of the HIT data model are presented. The model is based on the notion of function and uses the apparatus of the typed lambda-calculus which enables clean and transparent formulation of the data base concepts at various levels of description.

1. Introduction

At the moment, a number of data models has been published that reflect various aspects of data bases, or their proposed usage or their place in the hierarchy of levels ranging from the conceptual schema design of a data base to the rather physical work with the data. Nevertheless, the contemporary models are not general enough and cannot be succesfully used for all the levels of the description of data and the work with them.

In this paper, the basic concepts of the Homogeneous Integrated Type-oriented data model (HIT DM) [16] are presented. The model has been elaborated so as to be of use both for the conceptual schema design oriented to the use of natural language [12], and for the description of work with the data in a data base [9], [16], [11], [28]. In contrast with the approaches of e.g. [18], [4], [21], the main aim of HIT DM is to provide not only a functional semantics or a query language, but really a self-contained data model. HIT DM will be presented in an updated formalism which is based on the typed lambda-calculus (cf. e.g. [3], App.B) in a slight modification [31] inspired especially by the Simple Theory of Types [6] as modified in [24]. All the aspects of the link between the data model and the intensional analysis of the natural language [23] cannot be discussed here, the reader can consult [15] or [17].

2. Basic concepts

The main feature of the HIT DM is the use of functions for the description of the dependences between the data values (that, e.g., are to be stored in a data base). For the data description, the role of functions in HIT DM is similar to that of relations in the relational data model (RDM) [7]. For the work with functions, especially for their application and formation of new ones, the typed lambda-calculus is used in HIT DM in a way relational operations are used in RDM. Contrary to RDM, however, the use of functions can eliminate some difficulties connected with the work with "unnormalized relations"; moreover, the apparatus for the work with functions makes it possible to describe what happens in the external views, update operations or distributed data bases, and what, e.g., the application of integrity constraints means. In addition, the analysis of the natural language can be also based on functions [23]. This means that the same apparatus can be applied to the conceptual schema design of a data base (roughly speaking, the relationships in the E-R model [5] can be replaced by certain functions, cf. [12], in a similar way as relations were replaced by functions in the case of RDM). The ability of using the same apparatus both for the description of the data structures as well as their transformations and programs working over them, and for the description of the semantics of the initial notions from the modelled reality makes HIT DM to be really "homogeneous" - the same apparatus is used at various levels of description.

Two essential notions of HIT DM are <u>attributes</u> and <u>pseudo-attributes</u>. (It should be stressed that in HIT DM the notion of attribute is used in a different sense than in RDM.) To put it informally, attributes are the functions that one bears in mind when he uses the natural language. Take, e.g., the attribute "Salary". When we say

"The salary of Mr.Smith is $54000", (1)

this means that the function "Salary", applied to "Mr.Smith", gives the result "$54000", i.e.

"Salary"("Mr.Smith") = "$54000". (2)

However, the attributes in the natural language are not so simple because they are, in addition, parametrized by the state of the world (they are the so-called intensions [23]). The "Salary" includes the (empirical) dependency on the state-of-affairs, the so-called possible worlds [25]. Then, the meaning of (1) would be: take a particular possible world W (usually that which is believed to be the actual one, i.e. corresponding to the distribution of certain features realized in reality) and apply "Salary" to W; the result "Salary"(W) is such a funtion that having been applied to "Mr.Smith", it gives the result "$54000", i.e. instead of (2) we should write

("Salary"(W))("Mr.Smith") = "$54000".

A more sophisticated view of the semantics of the natural language can be achieved by the parametrization of the attributes (or intensions in the general theory) not only by the possible worlds but also by time moments. The details will not be discussed here, cf. [25].

When dealing with a particular data base, the data stored in it are rather fixed functions (or tables) rather than that kind of attributes parameterized by the state of the world. In a particular data base one deals with pseudo-attributes, cf. [15], [17]: concreted attributes for some particular corresponding state of the world (i.e. extensions of the former intensions). In HIT DM the pseudo-attributes play the role similar to the relations in RDM, or Codasyl-sets from the CODASYL report [8], while the attributes correspond to relationships and/or attributes from the E-R model.

For work with functions, it is useful to assign certain types to them, i.e. to connect a function with the domains of possible argument values and with the range of result values. For this purpose, a hierarchy of (functional) types over a certain set of elementary, non-functional, base types can be built. The elementary types will be the so-called sorts [17]; from the viewpoint of a data base designer or a data base user, sorts reflect the value domains in the same sense as the domains of attributes in RDM [13].

3. Formal apparatus

Let us have a base consisting of a finite number of denumerable domains of elementary values, the so-called sorts, one of them let be the set $\mathbf{B}$ (=Boolean) of the truth-values. The hierarchy of types over the base is inductively defined as follows:

(i) any member of the base is a type;

(ii) if T_1 and T_2 are types then also the domain of mappings from T_1 into T_2, denoted $(T_1 \to T_2)$, is a type (a functional type);

(iii) if $T_1, \dots, T_n$ are types then also their cartesian product (product domain), denoted $(T_1, \dots, T_n)$, is a type (a tuple type); in the case of n=1 we take (T_1) to be identical with T_1 and, moreover, we neglect embedded parentheses, i.e. we identify, e.g., $(T_1, (T_2, T_3))$ with (T_1, T_2, T_3) (we shall speak about the tuple types without embedded parentheses like, e.g., (T_1, T_2, T_3) as about normal types).

(iv) if T_1, T_2 are types then also their disjoint sum, denoted $(T_1 + T_2)$, is a type (a union type).

Note that we have supposed only unary functions in (ii). However, combining (ii) with (iii) we obtain also types of functions of any arity, e.g. $((T_1, \dots, T_n) \to T_{n+1})$, etc. We assume that every type includes certain member $\perp$, "undefined", of the appropriate type.

The possibility of forming union types (iv) enables to consider also subtypes since any system of subtypes clearly induces a corresponding division of the base types into smaller disjoint ones from which the former types can be obtained as union types.

Let us remark that we have introduced types corresponding only to simple values, functions and tuples, and that we have omitted types that would correspond to sets. The type of sets of members from T can be understood to be $(T \to \mathbf{B})$, because every set with members from T can be identified with its characteristic function from $(T \to \mathbf{B})$. The types of relations are clearly $((T_1, \dots, T_n) \to \mathbf{B})$, i.e. sets of tuples.

For the work with functions, we use the typed lambda-calculus slightly modified for the use of tuple types [31]. Terms of the calculus consist of variables, symbols (object or constant ones), each variable or symbol having some type, and the improper symbols. To each term of the calculus a certain type is assigned. The lambda-terms (or terms, for short) are defined as follows:

(i) Every symbol or variable is a term of the same type as the symbol or variable.

(ii) Let A be a term of type $(T_1 \to T_2)$ and B be a term of type T_1; then (AB), i.e. the application of A to B, is a <u>term</u> of the type $\mathbf{T}_2$.

(iii) Let A be a term of type $\mathbf{T}$ and $x_1, \ldots, x_n$ be mutually different variables of respective types $\mathbf{T}_1, \ldots, \mathbf{T}_n$; then $\lambda x_1 \ldots x_n(A)$, i.e. the $x_1, \ldots, x_n$--<u>abstraction</u> of A, is a <u>term</u> of the type $((\mathbf{T}_1, \ldots, \mathbf{T}_n)\text{->}\mathbf{T})$. (The abstraction <u>binds</u> the variables $x_1, \ldots, x_n$ in the body A of the abstraction; terms having all their variables bound are called <u>closed terms</u>.)

(iv) Let $A_1, \ldots, A_n$ be terms of respective types $\mathbf{T}_1, \ldots, \mathbf{T}_n$, $n\geqslant 1$, then $(A_1, \ldots, A_n)$, i.e. the <u>tuple construction</u>, is a term of the type $(\mathbf{T}_1, \ldots, \mathbf{T}_n)$.

(v) Let A be a term the normal type of which is $(\mathbf{T}_1, \ldots, \mathbf{T}_n)$; then $A_{(1)}, \ldots, A_{(n)}$ (where the subscripts are improper symbols!), i.e. for a fixed i, $1\leqslant i\leqslant n$, the <u>i-th projection</u> of A, are <u>terms</u> of the respective types $\mathbf{T}_1, \ldots, \mathbf{T}_n$. (Note that the normality assumption is necessary; otherwise the projections would be defined ambiguosly.)

(vi) Let A be a term of type $\mathbf{T}_i$, $1\leqslant i\leqslant 2$; then $i_{\mathbf{T}_i}^{(\mathbf{T}_1+\mathbf{T}_2)}A$, i.e. A <u>injected</u> into $(\mathbf{T}_1+\mathbf{T}_2)$, is a <u>term</u> of the type $(\mathbf{T}_1+\mathbf{T}_2)$.

(vii) Let x, y be variables of respective types $\mathbf{T}_1, \mathbf{T}_2$; A, B let be terms of type $\mathbf{T}$ not containing the variables y, x, respectively; then $\gamma xy(A,B)$, i.e. the <u>union abstraction</u>, is a <u>term</u> of type $((\mathbf{T}_1+\mathbf{T}_2)\text{->}\mathbf{T})$.

As a rule, we shall omit the outer pairs of parentheses in applications, i.e. we shall write AB or $A(B)$ instead of (AB) or $(A(B))$, whenever no confusion could arise.

Formally, the value of a lambda-term is defined as applying the semantic function $\mathcal{E}$ to the term and a particular interpretation. By an interpretation ϕ we shall understand a function that assigns to every variable and symbol of any type $\mathbf{T}$ some corresponding member of $\mathbf{T}$. (In fact, the distinction of variables and symbols has not any deeper sense than to enable a more friendly notation). From ϕ we can form interpretation that differs from ϕ only in assigning, e.g., to a symbol S the corresponding value $\mathcal{J}$, we shall denote such an interpretation by $[S\text{<-}\mathcal{J}]\phi$. In our calculus the semantic function $\mathcal{E}$ is defined by (terms of the formal lambda--calculus used as arguments of $\mathcal{E}$ are enclosed in $[\![\ ]\!]$):

$\mathcal{E}[\![S]\!]\phi=\phi(S)$ iff S is variable or symbol ;

$\mathcal{E}[\![(AB)]\!]\phi=(\mathcal{E}[\![A]\!]\phi)(\mathcal{E}[\![B]\!]\phi)$;

$\mathcal{E}[\![\lambda x_1 \ldots x_n(A)]\!]\phi=\lambda \underline{x}_1 \ldots \underline{x}_n(\mathcal{E}[\![A]\!][x_1\text{<-}\underline{x}_1, \ldots, x_n\text{<-}\underline{x}_n]\phi)$;

$\mathcal{E}[\![(A_1, \ldots, A_n)]\!]\phi=\text{n-tuple}(\mathcal{E}[\![A_1]\!]\phi, \ldots, \mathcal{E}[\![A_n]\!]\phi)$;

$\mathcal{E}[\![A_{(i)}]\!]\phi=\text{i-th}(\mathcal{E}[\![A]\!]\phi)$;

$\mathcal{E}[\![i_{\mathbf{T}_i}^{(\mathbf{T}_1+\mathbf{T}_2)}A]\!]\phi=\text{the-copy-of } \mathcal{E}[\![A]\!]\phi \text{ in } \mathbf{T}_i\text{-part of } (\mathbf{T}_1+\mathbf{T}_2)$;

$\mathcal{E}[\![\gamma xy(A,B)]\!]\phi=\lambda\underline{z}(\text{in-case } \underline{z}\in\mathbf{T}_1 : (\mathcal{E}[\![\lambda x(A)]\!]\phi)\underline{z};\ \underline{z}\in\mathbf{T}_2 : (\mathcal{E}[\![\lambda y(B)]\!]\phi)\underline{z})$.

For the defining notation (on the right-hand side) we use a notation that represents LAMBDA [22] or constructions [24], both with added constructs for work with tuple types; or just a slightly modified calculus from [20,ch.IV]. We use underlined variables in the defining notation to distinguish them from the variables of the formal lambda-calculus. The structural similarity of the formal language of lambda-terms and the notation for the members of types presents no difficulty because particularly the similarity allows more transparent insight into the (data base) subject.

Over the lambda-terms it is possible to build up an equational theory describing the natural transformations of lambda-terms, particularly the so-called β-conversion, [3], [31]. We can choose special constant symbols and interpret them in such a way in which they can play in our calculus the same role as the logical connectives and quantifiers play in the traditional logic [23]: The traditional logical connectives can be replaced by constant symbols $\wedge$, $\vee$, $\supset$, $\equiv$ of the type $((\mathbf{B},\mathbf{B})\text{->}\mathbf{B})$ and $\neg$ of the type $(\mathbf{B}\text{->}\mathbf{B})$. The identity test, $=$, is in fact a constant symbol of a type $((\mathbf{T},\mathbf{T})\text{->}\mathbf{B})$, for every type $\mathbf{T}$. Expressions like $\forall x(A)$, $\exists x(A)$, or $\iota x(A)$ (the last read "the only x such that A holds for this value of x), with the variable x of type $\mathbf{T}$, can be replaced by $\Pi(\lambda x(A))$, $\Sigma(\lambda x(A))$ or $I(\lambda x(A))$, respectively, where the constant symbols Π, Σ and I of the respective types $((\mathbf{T}\text{->}\mathbf{B})\text{->}\mathbf{B})$, $((\mathbf{T}\text{->}\mathbf{B})\text{->}\mathbf{B})$ and $((\mathbf{T}\text{->}\mathbf{B})\text{->}\mathbf{T})$ are interpreted by the following functions:

$\phi(\Pi)$ ... returns true iff its argument is the set containing just all the members of $\mathbf{T}$;

$\phi(\Sigma)$ ... returns true iff its argument is a non-empty set;

$\phi(I)$ ... returns the only member of its argument (or the value $\perp$ if the argument contains any other number of members than 1).

The interpretation of constant symbols can be fixed stating an appropriate equational theory for the constant symbols [29]. In particular, all the interpretations considered in the following are assumed to assign proper objects to constant symbols.

Note that for every type $\mathbf{T}$ one needs an extra triple Π, Σ, I, like in the case of $=$. For the notational purposes, we shall use the traditional form of notation instead of the functional one for the logical constant symbols

For the sorts, it is useful to have special constant symbols enabling to name the elements of the sorts; the constant symbol that is interpreted by a member t of sort $\mathbf{T}$ is denoted by 't' (of the type $\mathbf{T}$). Similarly, for every sort $\mathbf{T}$ there is a natural ordering of the members of the sort, e.g. the sort of natural numbers, etc., we assume that we have the corresponding symbol < of the type $((\mathbf{T},\mathbf{T})\text{->}\mathbf{B})$ interpreted by the test for "less than" (in the ordering); again, we can use infix notation for these symbols.

The set of the constant symbols mentioned above will be denoted by $\mathfrak{L}$.

4. Data bases

Let us state more precisely how to define a data base using our formal means. First, one should express what a part of realiy he wants to deal with. This is managed via specifying the data base concept, which is a tuple of attributes of interest. The data base concept materializes in the process of the conceptual design into a data base conceptual schema ([12, [10] deal with the methods of the conceptual design using particularly HIT DM). Then all the logically possible functions, by which extensions of these attributes can be interpreted, must be restricted to those admissible with respect to our knowledge of the properties of the actual world. After such a restriction (usually performed in the designer's mind), one is able to state the "reasonable" domains of the pseudo-attributes that can be realized in reality. These domains become sorts and form (possibly with the added sort $\mathbf{B}$) the base of sorts [13]. The base of sorts for a particular data base results from the attributes of interest (of the data base concept); therefore we can restrict the types of pseudo-attributes to the simple types which are either (suppose $\mathbf{T}_1,\dots,\mathbf{T}_n$, $\mathbf{S}_1,\dots,\mathbf{S}_m$ are sorts):

(i) $((\mathbf{T}_1,\dots,\mathbf{T}_n)\text{->}(\mathbf{S}_1,\dots,\mathbf{S}_m))$, or

(ii) $((\mathbf{T}_1,\dots,\mathbf{T}_n)\text{->}((\mathbf{S}_1,\dots,\mathbf{S}_m)\text{->}\mathbf{B}))$.

i.e. to functions whose result values are (i) tuples (i.e., equivalently, a tuple of m functions with simple result from one sort), or (ii) relations (a special case are sets).

To form a data base schema, let us assign an attribute identifier, which will be an object symbol of a simple type, to every attribute of the data base concept. The data base schema is then the term

$$(A_1,\dots,A_n) \quad ,$$

where $A_1,\dots,A_n$ are the attribute identifiers; i.e. the data base schema is a tuple of attribute identifiers.

Note that, given a data base schema S of a normal type $(\mathbf{T}_1,\dots,\mathbf{T}_n)$, we can express its decomposition into the attribute symbols either by saying $S=(A_1,\dots,A_n)$ or using projections, i.e. $S=(S_{(1)},\dots,S_{(n)})$. The latter can be especially used when one deals with transformed schemata.

A data base schema becomes a data base when its attribute identifiers are filled with data, i.e. when there is an interpretation of the attribute symbols. The interpretation will be called the data base state.

Example 1: Let us introduce a simple example that occurs in several papers. Let the base of sorts consist of (besides the set $\mathbf{B}$ of the truth-values): String, Num, Sal, Floor, Emp, Dept, Comp, Itm, Typ, Addr meaning the sorts of names, quantities, salaries, floors, employees, departments, companies, items, types of items and addresses of companies, respectively. Choose the following identifiers for our attributes of interest (read slashes as "of type"):

NE	/ (Emp->String)	.... name of an employee;
SE	/ (Emp->Sal)	.... salary of an employee;
ME	/ (Emp->Emp)	.... manager of an employee;
DE	/ (Emp->Dept)	.... department of an employee;
ND	/ (Dept->String)	.... name of a department;
QDSI	/ ((Dept,Itm)->Num)	quantity in which a department sells an item;
QSCID	/ ((Comp,Itm,Dept)->Num)	...... quantity in which a company supplies an item to a department;
NI	/ (Itm->String) .	name of an item;
AC	/ (Comp->Addr)	.... address of a company;
LD	/ (Dept->Floor)	.... location of a department;
TI	/ (Itm->Typ)	type of an item.

The corresponding data base schema is

$$D = (NE,SE,ME,DE,ND,QCSI,QCSID,NI,AC,LD,TI). \quad (3)$$

⌘

Let us remark that a data base schema composed of functions need not dismay anyone who is familiar with RDM: The natural understanding of the dependences between the data describing reality is really functional and the functionality appears in relational schemata like a stowaway in the form of, e.g., keys of relations.

However, notwithstanding the fact that relations do not seem to be as natural as simple functions, relations are a special kind of attributes because $((\mathbf{T}_1,\dots,\mathbf{T}_n)\text{->}\mathbf{B})$ is exactly the type of relations over sorts $\mathbf{T}_1,\dots,\mathbf{T}_n$. Therefore, one can regard RDM as a special case of HIT DM.

5. Retrieval

By a retrieval operation we understand a function that transforms a given data base into a requested answer. The answer is a member of some appropriate type over the base of sorts. Usually, the type of the answer is some sort, particularly the sort $\mathbf{B}$ in the case of yes/no queries, or a type $(\mathbf{T}\text{->}\mathbf{B})$, for some sort $\mathbf{T}$ (i.e. type of a set of objects of the sort $\mathbf{T}$). More complicated cases are also possible and they can express even structures of complex answers, e.g. structured tables, using nested sets, relations or functions.

We shall use the lambda-calculus for the definition of retrieval operations. Naturally, much more friendly user-oriented means can be taken into account [16], [11], [19], [27]; nevertheless, the lambda-calculus represents some kind of "naked semantics" for them.

Let us have a set $\mathfrak{C}$ of constant symbols that contains the set $\mathfrak{L}$ of "logical" symbols (see above, §3), i.e. $\mathfrak{L}\subseteq\mathfrak{C}$, and let S be a data base schema of type $\mathbf{S}$. Then the retrieval operation of class $\mathfrak{C}$ will be such a closed term of type $(\mathbf{S}\text{->}\mathbf{T})$ that does not contain any constant symbol other that the symbols from . It is not difficult to show that the retrieval operations of the class $\mathfrak{L}$ correspond to the relational algebra or the relational calculus of RDM [19].

An enrichment of $\mathfrak{L}$ by new functions, e.g. aggregating functions as "count", "min", "max", "sum", "aver" or arithmetic functions as $+,-,\times,/$, etc., produces no formal complications and enables us to obtain richer classes of retrieval operations that correspond to commonly used user-oriented retrieval languages.

In general, for a given data base schema S of type $\mathbf{S}$, terms defining retrieval operations can be written as

$$\lambda a_1 \ldots a_n (R) \ , \qquad (4)$$

where the variables $a_1,\ldots,a_n$ are of the same types as attribute identifiers $A_1,\ldots,A_n$, provided that $S = (A_1,\ldots,A_n)$. More usual it is to write such a query in the form

$$R[a_1\text{<-}A_1,\ldots,a_n\text{<-}A_n] \qquad (5)$$

(brackets denote formal substitution), because in (5) one need not take care of the particular structure of the data base schema as (5) contains only the attribute identifiers relevant to the retrieval. Of course, (5) is equivalent to the retrieval operation (4) applied to the given data base schema S, i.e. (β-conversion, cf. e.g. [31]):

$$R[a_1\text{<-}A_1,\ldots,a_n\text{<-}A_n]=(\lambda a_1\ldots a_n(R))(S).$$

In such a case, given a data base state ϕ, the answer is given by

$$\mathfrak{E}[\![R[a_1\text{<-}A_1,\ldots,a_n\text{<-}A_n]]\!]\phi \ .$$

Example 2: Take the data base schema D from (3) and a query to the data base:

"Find the (names of the) departments where all the employees earn less then their manager."

One can easily check that this query is answered by (n, d, e are variables of the respective types String, Dept, Emp):

$$\lambda n(\exists d(ND(d)=n\wedge\forall e(DE(e)=d\supset \supset SE(ME(e))>SE(e)))) \ . \qquad (6)$$

Note that the type of this term is $(\underline{\text{String}}\text{->}\mathbf{B})$, i.e. that of sets of names.

The set that answers our query, according to a given data base state ϕ, is

$$\mathfrak{E}[\![\lambda n(\exists d(ND(d)=n\wedge\forall e(DE(e)=d\supset \supset SE(ME(e))>SE(e))))]\!]\phi \ .$$

Let us stress that the formal language of lambda-terms makes it possible to express the semantics of operations over data bases in a unified way and without a barrier of syntax. Although it was not meant to provide a user-oriented query language, the very form of lambda-terms should not be unfamiliar to the data base people. To realize it, compare the form of (6) with the examples of the same query (only slightly modified by the identification of the descriptive sorts with the object sorts, cf. [9] or [10]) expressed in common relational query languages in [14] (one could even abbreviate $\lambda n(\ldots)$ by $\{n:\ldots\}$).
⌘

6. External views

The data base schema represents a kind of a global view of both data stored in a data base and their structure. However, from the viewpoint of a particular user, it is desirable to be able to see the data from his own view. For this purpose, the user uses his own view schema derived from a data base schema using a transformation called (external) view.

View schema is, like the data base schema, a tuple of attribute identifiers. However, dealing with view schema, the attribute identifiers are no longer symbols only but generally lambda-terms (with types of the form either (i) or (ii) from §4) that contain, besides bound variables and constant symbols, at most the attribute identifiers of the source data base schema.

A view that gives rise to a view schema is lambda-term U of type $(\mathbf{S}\text{->}\mathbf{V})$, where $\mathbf{S}$ is the type of the (source) data base schema S and $\mathbf{V}$ is the type of the (target) view schema V. Then we have

$$V = U(S) \quad ,$$

and the attribute identifiers of the view schema V are $(U(S))_{(1)},\ldots,(U(S))_{(n)}$, for V of a normal type $(\mathbf{T}_1,\ldots,\mathbf{T}_n)$.

View schemata contain no more symbols (other than the constant ones or bound variables) than the original data base schema. The state of a view data base (the interpretation that assigns data to attribute identifiers) is always identical with that of the source one.

Any data base schema can be also considered to be a view schema given by $U=\lambda a(a)$ that defines the identity mapping. It would be useful to retell the story concerning retrieval operations and to formulate retrievals for schemata whose attribute identifiers are, in general, lambda-terms. Clearly, this presents no difficulties. Therefore, one can use schema with the meaning of both data base schema and view schema, especially in the context of retrieval and view operations.

Views can provide suitable representations of, e.g., the transformers of the ANSI/X3/SPARC schema [1]. If we choose suitable views that do not "forget" any information, to every view U we have a corresponding inverse view U^{-1} such that for a given schema S, S is equivalent with both $U^{-1}(U(S))$ and $U(U^{-1}(S))$ (cf., e.g., [9]). In this case, one may choose as the very data base schema any of I-, C- or E- schemata. (However, the I-schema is probably the best choice in a real system because then the (abstract) data base state has direct (concrete) explanation as the state of the physical files.)

Of course, one may also imagine a view schema that reflects the corresponding schema in any other data model the data structures of which are expressible using the HIT-like attributes. Particularly, this is true for RDM.

Example 3: Let us show the view R that maps the schema D from (3) into the relational schema with relations:

EMP / ((String,Sal,String,String)->$\mathbf{B}$);
SALES / ((String,String,Num)->$\mathbf{B}$);
SUPPLY / ((Comp,String,Itm,Num)->$\mathbf{B}$);
SUPPLIER / ((Comp,Addr)->$\mathbf{B}$);
LOC / ((String,Loc)->$\mathbf{B}$);
CLASS / ((String,Typ)->$\mathbf{B}$).

Note that the multiple String's in these relations were caused by our initial choice of sorts in Example 1. If we chose different sorts of names for employees, departments and items, the types of the relations would look more naturally. The corresponding view R will be (the variable a is of the same type as D, and variables n,s,o,p,e,m,d,q,i,c, r,l,v,t are of the respective types String,Sal, String,String, Emp,Emp,Dept, String,Itm,Comp,Addr,Loc,Num,Typ):

$R=\lambda a((\lambda nsop(\exists e(a_{(1)}(e)=n \wedge a_{(2)}(e)=s \wedge$
$\exists m(a_{(1)}(m)=o \wedge a_{(3)}(e)=m) \wedge \exists d(a_{(5)}(d)=p$
$a_{(4)}(e)=d),\ \lambda pqv(\exists di(a_{(5)}(d)=p \wedge a_{(8)}(i)=q \wedge$
$a_{(6)}(d,i)=v)),\ \lambda cpqv(\exists di(a_{(5)}(d)=p \wedge$
$a_{(8)}(i)=q \wedge a_{(7)}(c,d,i)=v)),\ \lambda cr(a_{(9)}(c)=r),$
$\lambda pl(\exists d(a_{(5)}(d)=p \wedge a_{(10)}(d)=e)),$
$\lambda qt(\exists i(a_{(8)}(i)=q \wedge a_{(11)}(i)=t)))$.

Denoting

EMP = $\lambda nsop(\exists e(NE(e)=n \wedge SE(e)=s \wedge$
$\exists m(NE(m)=o \wedge ME(e)=m) \wedge \exists d(ND(d)=p \wedge$
$DE(e)=d)$;
SALES = $\lambda pqv(\exists di(ND(d)=p \wedge NI(i)=q \wedge$
$QDSI(d,i)=v))$;
SUPPLY = $\lambda cpqv(\exists di(ND(d)=p \wedge NI(i)=q \wedge$
$QCSID(c,i,d)=v))$;
SUPPLIER = $\lambda cr(AC(c)=r)$;
LOC = $\lambda pl(\exists d(ND(d)=p \wedge LD(d)=l))$;
CLASS = $\lambda qt(\exists i(NI(i)=q \wedge TI(i)=t))$;

the corresponding (relational) view schema R(D) is equivalent to R(D) = = (EMP,SALES,SUPPLY,SUPPLIER,LOC,CLASS) with the semantics exactly fitting the schema that one would express in RDM.
*

Slightly generalizing views to mappings from any number of source data base schemata, it is easy to describe also schemata of distributed data bases, cf. [32], or [27].

7. Updates

So far we have considered only transformations of data base schemata under a fixed data base state, i.e. uder a fixed interpretation. However, it is also necessary to be able to change the data base state, i.e. to transform it from one state into another performing updates of a data base: by an update we shall understand a mapping U from the interpretations (of symbols from a given data base schema) into themselves.

Consider a data base schema S of a type $\mathbf{S}$; $S=(A_1,\ldots,A_n)$. Let us have a term U of a type $(\mathbf{S}\to\mathbf{S})$ containing only bound variables or constant symbols. By the update defined by U (shortly update only) we shall understand such a mapping $\mathfrak{U}$ from interpretations into interpretations that assigns to interpretation ϕ corresponding result $[A_1\leftarrow\text{first}(\mathcal{E}[\![US]\!]\phi), \ldots, A_n\leftarrow n\text{-th}(\mathcal{E}[\![US]\!]\phi)]\phi$, i.e.

$$\mathfrak{U}:\ \phi\longmapsto[A_1\leftarrow\text{first}(\mathcal{E}[\![US]\!]\phi),\ldots,A_n\leftarrow n\text{-th}(\mathcal{E}[\![US]\!]\phi)]\phi. \quad (7)$$

The lambda-term U is called the defining term of the update $\mathfrak{U}$.

Defining terms of updates play an important role in the treatment of update operations and especially in the context with integrity constraints (cf. [30]). It is easy to show (using the definition of $\mathcal{E}[\![\ ]\!]$) that every defining term U of an update $\mathfrak{U}$ represents in fact a view such that, given a fixed data base state, the data base corresponding to the view schema given by the view U is exactly the same as the updated data base with the original data base schema, i.e. for any data base schema S, state ϕ and update $\mathcal{U}$ it holds

$$\mathcal{E}[\![US]\!]\phi = \mathcal{E}[\![S]\!](\mathcal{U}\phi) \ , \qquad (8)$$

where U is the defining term of $\mathcal{U}$.

The impact of the equation (8) consists in the way in which it determines a particular update: First, one defines a view that "models" the result of the update. Then, using (8), such a view is the defining term of the considered update; thus, using (7), the update is fully described. Therefore, according to (8), one is able to study updates using the same techniques as for the study of views.

Naturally, it would be awkward if every new update required to do the corresponding derivation separately. Instead, it is more natural to introduce a notion of update operations, which are functions assigning to given parameters a corresponding update [30], [26]. The update operations are treated like updates (with only slight changes, e.g. defining term is of the type $((\mathbf{P}_1,\dots,\mathbf{P}_n)\text{->}(\mathbf{S}\text{->}\mathbf{S}))$, where $\mathbf{P}_1,\dots,\mathbf{P}_n$ are types of the parameters of the update operation). Properties similar to (7), (8) are also satisfied.

When one deals with updates, it is natural to introduce compositions of updates. It is easy to show that, having two updates $\mathcal{U}$, $\mathcal{V}$ with the corresponding defining terms U, V, the term

$$U\circ V = \lambda a(U(V(a)))$$

is the defining term of the composed update $\mathcal{U}\circ\mathcal{V}$.

Having a set $\mathcal{W}$ of updates, we say that $\mathcal{W}$ is semicomplete iff $\forall\, \mathcal{U}, \mathcal{V}\in\mathcal{W}$: $\mathcal{U}\circ\mathcal{V}\in\mathcal{W}$. Moreover, if $\mathcal{W}$ is semicomplete and $\forall\mathcal{U}\,\exists\mathcal{V}$: $\mathcal{U}\circ\mathcal{V} = \mathcal{I}$ (where $\mathcal{I}$ is the identity mapping), we say that $\mathcal{W}$ is complete [2].

Similarly, both completeness and semicompleteness can be analogically defined for update operations.

Using views, it is easily possible to formalize in a quite similar way also view updates (cf. [32]) via defining lambda-terms called update translators. They behave like translators from [2].

8. Integrity constraints

No data model seriously dealing with updates can ignore the fact that admissible data in a data base must fulfil some constraints, the consistency constraints, that follow from our knowledge of reasonable links among the data describing reality. In fact, consistency constraints are given through a specification (using some statements of natural language declaring properties of admissible (i.e. not only logically possible) reality; they should take part in the design of the data base conceptual schema together with the data base concept) of a set of admissible data base states, the data base space. The data base space is given by lambda-term C called the consistency checker [30] of type $(\mathbf{S}\text{->}\mathbf{B})$, where the corresponding data base schema S is of the type $\mathbf{S}$. Then, the corresponding data base space is the set

$$Sp = \lambda\phi(\mathcal{E}[\![C(S)]\!]\phi) \ ,$$

i.e. the set of all the interpretations ϕ such that $\mathcal{E}[\![C(S)]\!]\phi$ is "true".

As the first approximation, one can use the integrity checker as follows:

(i) perform update $\mathcal{U}$ to the data base $\mathcal{E}[\![S]\!]\phi$;

(ii) then apply C to the resulting data base, i.e. perform the yes/no query $\mathcal{E}[\![C(S)]\!](\mathcal{U}\phi)$;

(iii) if the result of (ii) is "true" then accept the update (i) else resume (i) and return the data base into the initial state ϕ .

This schema can be, however, substantially improved. First, according to (8), one can perform the test (iii) before performing the update because it is sufficient to perform the yes/no query

$$\mathcal{E}[\![C(U(S))]\!]\phi \ , \qquad (9)$$

where U is the defining term of the update $\mathcal{U}$. In that case, an inconsistent update can be resumed even before one tries to perform it.

Even now, the checking of admissibility of an update using (9) is not the best solution. According to a given update (or update operation) one can imagine such a test that does not check all the data in a data base for consistency (as (9) in fact does) but that tests only the data that are somehow connected with the particular update. An optimal test is realized using the so-called integrity constraint I (for a particular update given by U), which is a term of type $(\mathbf{S}\text{->}\mathbf{B})$ such that

$$\forall\phi{:}\ \mathcal{E}[\![C(S)]\!]\phi=\text{true} \Rightarrow \mathcal{E}[\![C(U(S))]\!]\phi = \mathcal{E}[\![I(S)]\!]\phi \ . \qquad (10)$$

The integrity constraint I is not defined by (10) unambiguously, the particular choice depends on a suitable optimization criterion which does not, however, come from the calculus itself.

In practice, it is possible to define the integrity constraint as such a term I for which

$$I(S) = C(U(S)) \qquad (11)$$

in the theory enriched by the axiom:

$$C(S) = \text{'true'}$$

Particularly, if C and U contains only such symbols for which it is possible to define such an equational theory in which $\forall\phi$: $\mathcal{E}[\![A]\!]\phi = \mathcal{E}[\![B]\!]\phi$ iff A = B, then (11) is equivalent with (10). In general, some I's obeying (10) may not satisfy (11); from the viewpoint of the practically implementable consistency constraints, however, it should not be important.

For a more detailed treatment of the topic, cf. [30], [28] or [29].

9. HIT DM and implementations

Although we have proposed the ideas of HIT DM on the level of an abstract model, it is applicable also to the work on the internal level of a DBMS. (Pseudo-)attributes correspond to virtual data files accessed using keys: function arguments represent the keys while the results represent the items of file records ([9] uses special types for the internal schema attribute identifiers).

Lambda-terms defining retrievals can be directly translated into a (procedural programming language using ordinary commands for data manipulation; thanks to the simple apparatus of conversions, it is advantageous to do various optimizations on the level of lambda -terms (cf. [9] for more details).

10. Conclusion

We have described the basic framework of the HIT data model. It represents a model that is theoretically at least as well-grouded as the relational one; however, it is able to be successfully used in a wider spectrum of applications. Over and above, the formal apparatus of the model enables us to achieve a unified description of various levels of data base systems, from the design of a conceptual schema to the semantics of physical implementation.

Acknowledgments: I would like to thank to all the authors of [16], especially to Prof. Pavel Materna, for encouraging discussions of the formal framework.

References:

[1] ANSI/X3/SPARC, Interim report 75-02-08, FDT Vol. 7, 1975.
[2] Bancilhon,F.-Spyratos,N.: Update semantics of relational views, ACM Vol.6, 1981.
[3] Barendregt,H.: The lambda calculus. North-Holland, 1981.
[4] Buneman,O.P.-Frankel,R.E.: FQL - a functional query language, Proc. ACM SIGMOD, 1979.
[5] Chen,P.P.S.: The entity-relationship model, ACM TODS, Vol. 1, 1976.
[6] Church,A.: A formulation of the simple theory of types, J. Symb. Logic, Vol.5, 1940.
[7] Codd,E.F.: A relational model of data for large shared data banks, Comm. ACM, Vol.13, 1971.
[8] Report of DBTG CODASYL, ACM, 1971.
[9] Felix,O.-Zlatuška,J.: Transforming external queries into internal operations with data using HIT DM, 8th Int. Sem. on DBMS, Oct. 1985
[10] Krejčí,F.: The HIT metodology for a complicated conceptual data structure design, to appear.
[11] Krejčí,F.-Kohout,M.: FUL - the functional user-oriented language for the HIT DM, to appear.
[12] Krejčí,F.-Zlatuška,J.: Natural language and its role in designing and performing data bases, to appear.
[13] Krejčí,F.-Zlatuška,J.: Sortalization for the HIT DM, forthcoming.
[14] Lacroix,M.-Pirotte,A.: Example queries in relational languages, 1977
[15] Materna,P.: Theory of types and data description, Kybernetika, Vol.14, 1978.
[16] Materna,P.- Krejčí,F.- Zlatuška,J.--Pokorný,J.-Felix,O.: HIT - data base model, Proc. SOFSEM'81, 1981 (in Czech).
[17] Materna,P.-Pokorný,J.: Applying simple theory of types to data bases, Inf. Systems, Vol.6, 1981.
[18] Neuhold,E.J.-Olnhoff,Th.: Building data base management systems through formal specification, in LNCS 107, 1981.
[19] Pokorný,J.: Functional approach to conceptual modelling, thesis, 1984 (in Czech).
[20] Scott,D.S.: Lectures on a mathematical theory of computation, Oxford Univ., 1981.
[21] Shipman,D.W.: The functional data model and the data language DAPLEX, ACM TODS, Vol. 6, 1981.
[22] Stoy,J.E.: Denotational semantics: the Scott-Strachey approach to programming language theory, MIT 1977
[23] Tichý,P.: Intensional logic, Univ. of Otago, Ms., 1976.
[24] Tichý,P.: Foundations of the partial type theory, Reports on Math. Logic, Vol.14, 1982.
[25] Tichý,P.: On the limitations of the logical space, to appear.
[26] Zlatuška,J.: External views and updates in HIT DM, Proc. SOFSEM'82, 1982 (in Czech).
[27] Zlatuška,J.: The HIT data base model, Univ. of Brno, Ms., 1982.
[28] Zlatuška,J.: The data base model HIT, TU Brno, 1983 (in Czech).
[29] Zlatuška,J.: Integrity constraints in HIT DM, Proc. SOFSEM'83, 1983 (in Czech).
[30] Zlatuška,J.: Keeping integrity in HIT DM, Scripta Fac. Sci. Nat. Univ. Brno, Vol. 15, 1985.
[31] Zlatuška,J.: Normal forms in the typed lambda-calculus with tuple types, Kybernetika, 1985, in press.
[32] Zlatuška,J.: HIT data model, 7th Int. Sem. on DBMS, Varna, 1984.