Proc. Int. Joint Conf. on Theory and Practice of Software Development (TAPSOFT'89), Josep Diaz, Fernando Orejas (Eds.), Springer Berlin, LNCS 351, pp. 259-274 (1989)

Algebraization and Integrity Constraints for an Extended Entity-Relationship Approach

martin gogolla

Technische Universität Braunschweig

Informatik / Datenbanken

Postfach 3329

D-3300 Braunschweig

Federal Republic of Germany

e-mail: gogolla@infbs.uucp or gogolla@dbsinf6.bitnet

Abstract: An extended entity-relationship model concentrating nearly all concepts of known "semantic" data models and especially allowing arbitrary user defined data types is introduced. The semantics of the model is described purely in algebraic terms mainly based on the notions of signature, algebra and extension. On this basis a calculus making intensive use of abstract data types is defined and employed for the formulation of typical integrity constaints like functional restrictions and key specifications.

Keywords: Theory of data bases, data model, entity-relationship model, formal semantics, calculus, abstract data type, aggregate function, relational completeness, integrity constraint.

1. Introduction

Among the different steps for the design of a database the conceptual design plays a mayor role [TF82, Ce83]. Here all requirements of later database users are collected and described in a formal way. Many authors (among them [Ch76]) agree that the Entity-Relationship model is the most adequate data model to be used in this phase of database design. Quite a number of ER languages and ER algebras [PS85] (where the notion algebra is used analogously to relational algebra) have been proposed, but up to now nearly no work has been done in order to define an ER calculus (again, analogously to relational calculi). It is also rather an exception [Su87, Bü87] in the area of databases that languages have a formal semantics, more common is the definition by examples. Special interest in the formal description of database (andespecially ER) languages has to be paid to aggregate functions like this has been done for the relational approach [Kl82, ÖÖM87].

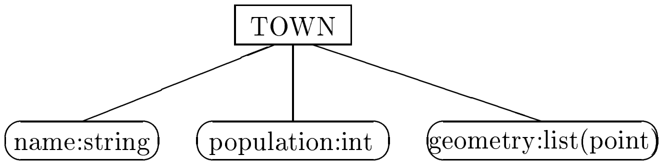
On the other hand, some papers have tried to combine ideas developed in the field of algebraic specification (see for instance [EM85]) with relational database design [EKW78, DMW82]. Also the modal systems of algebras as proposed in [GMS83, KMS85] supported the conceptual modelling by algebraic tools. Algebraic techniques were also successfully employed for the description of certain standard "universes" in connection with key specifications [EDG86,SSE 87].

This paper tries to bridge the gap between the entity-relationship and the algebraic specification communities. We propose an extended ER model having an algebraic semantics and use (a polished version of) our extended ER calculus [HG88] to formulate typical integrity constraints (to be used in the conceptual design). Let us finally remark that our calculus is not restricted to the ER model, it can be applied to other approaches [Sh81, JS82, SS86] as well.

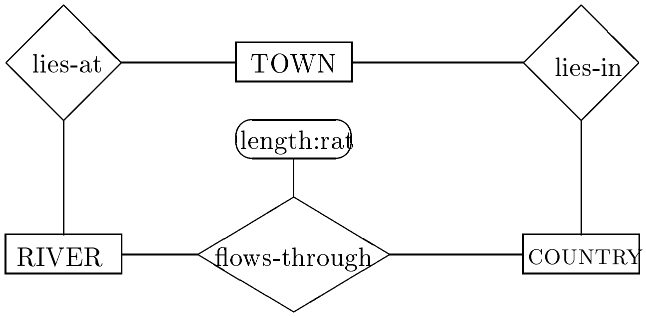
The paper is organized as follows: Chapter 2 gives an informal introduction into the data model and the calculus by means of an example. Chapter 3 formally defines the model and pointsout how to formulate it in algebraic terms. In chapter 4 the calculus is defined and it is applied in chapter 5 to formulate integrity constraints.

2. The Basic Idea

Before we explain the formal details of our approach we point out the basic ideas by means of an example. We consider a simple geo-scientific database where information about towns, countries and rivers has to be stored. First, we have entity types which are described by attributes returning values of given data types (possibly set-, bag- or list-valued), e.g., a TOWN has a name, a population and a geometry.

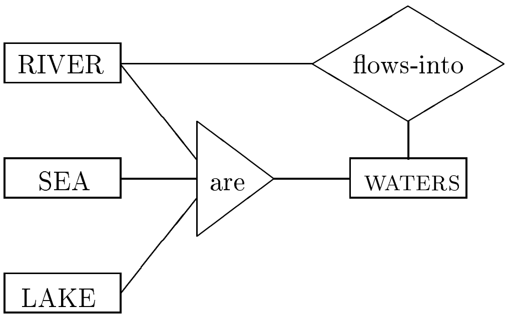


Secondly, relationships can exist between entity types, e.g., RIVERS flow through COUNTRIES, TOWNS lie at RIVERS and lie in COUNTRIES. These relationships can also have attributes.



Thirdly, entity types can have other entity types as components, e.g., a TOWN has as component the set of its DISTRICTS and each district has as component the set of STREETS lying in the district.

Last, but not least,entity types can be constructed from other entity types, e.g., an entity of type WATERS is constructed from an entity of type RIVER, SEA or LAKE. RIVER, SEA and LAKE are the input types of the construction, WATERS the output type.



Due to space limitations only a short sketch of our model can be given. Full motivation of all concepts can be found in [HG 88]. For an entity-relationship schema (as introduced above) we shall define a signature (in the sense of abstract data type theory) and a database state for such a schema will then be an algebra for this signature (with carrier sets, functions and relations).

We will also define a calculus for such entity-relationship schemas which can be employed to express queries and integrity constraints. The calculus especially distinguishes between sets and bags (multisets). Therefore it is well suited to formulate aggregation properties not expressible for instance in the "classical" relational tuple or domain calculus [Ma 83]. For example the query “Give me for each country its name and the average of the population of its towns” is formulated as

-[name(c) , AVG -[population(t) | t:TOWN $\wedge$ lies-in(t,c)]- | c:COUNTRY]-

Terms of the form -[...]- are bag-valued (retaining duplicates). Please notice, this is essential for the calculation of the average in the subquery, if two towns have the same population. It is also possible to restrict variables to the finite set of all stored values: Assume ‘government’ is an attribute for COUNTRY of sort string (for instance “socialistic”, “democratic”, etc.).Then the query “Give me every (stored) form of government and for every (stored) form of government the sum of country populations having this form” will be expressed as

-[g , SUM -[population(c) | c:COUNTRY $\wedge$ government(c) = g]- | g : BTS -[government(c) | c:COUNTRY]-]-

The standard function BTS converts a Bag To a Set. The calculus is also employed to formulate integrity constraints. For example, relationships can be required to be functiona:

($\forall$ fi, fi’ : flows-into) fi.RIVER = fi’.RIVER $\Rightarrow$ fi.WATERS = fi’.WATERS

This formula means that flows-into is functional from RIVER to WATERS, i.e., a river flows into (at most) one water. Another application is the specification of key attributes, key components and key relations:

name(r) $\neq$ name(r’) $\vee$ ($\exists$ c : COUNTRY) flows-through(r,c) xor flows-through(r’,c)

This formula, where xor stands for the exclusive or, expresses that a river is identified by its name and the countries through which it flows: {name, flows-through} is a key for RIVER. In other words, two different rivers must have different names or (if their names coincide) there must be a country such that one river flows through this country and the other one not: the rivers have to be observably inequivalent with respect to “name” or “flows-through”. Let us finally mention the possibility of cardinality constraints :

($\forall$ d : DISTRICT) 50 $\leq$ CNT(streets(d)) $\wedge$ CNT(streets(d)) $\leq$ 100

The above line (CNT stands for COUNT) says that districts have at least 50 and at most 100 streets.

3. Algebraization of the Extended Entity-Relationship Model

3.1 Axiomatic conventions: Let |SET| denote the class of sets, |FISET| the class of finite sets, |FUN| the class of total functions and |REL| the class of relations. There are the obvious inclusions |FISET| $\subseteq$ |SET| and |FUN| $\subseteq$ |REL| $\subseteq$ |SET|. Assume sets S, S_1, ..., $S_n \in$ |SET| are given. Then F(S) denotes the restriction of the powerset P(S) of S to finite sets, S^* the set of finite lists over S, S^+ the set of finite non-empty lists over S, and S_1 x ... x S_n the cartesian product of the sets S_1,...,S_n. The set of finite multisets or bags over S is given by B(S). A bag can be considered as a (finite) set S together with a counting function occur : S $\rightarrow \mathcal{N}$, giving for each element the number of occurrences in the bag.

Finite sets are written as $\{c_1,...,c_n\}$, lists as $<c_1,...,c_n>$, elements of the cartesian product as $(c_1,...,c_n)$, and bags as $\{\{c_1,...,c_n\}\}$. For a set $\{c_1,...,c_n\}$, i$\neq$j implies $c_i =/ c_j$. But this is not necessarily true for bags: If we have a bag $\{\{c_1,...,c_n\}\}$ with occur(c)=k, this implies that there are k distinct indices $i_1,...,i_k \in$ 1..n with c_{i_j} =c for j$\in$1..k. □

3.2 Definition: (data signature)

The syntax of a data signature DS is given by

- the sets DATA, OPNS, PRED $\in$ |FISET|,

- a function source: OPNS $\rightarrow$ DATA*,

- a function destination: OPNS $\rightarrow$ DATA, and

- a function arguments: PRED $\rightarrow$ DATA$^+$.

If $\sigma \in$OPNS, source(σ) = <d_1, ..., d_n >, and destination(σ)=d, this is notated as σ: d_1 x ... x d_n $\rightarrow$d. If $\pi \in$PRED with arguments(π) = <d_1, ..., d_n >, this is notated as π: d_1 x ... x d_n.

The semantics of a data signature DS is given by

- a function μ[DATA] : DATA $\rightarrow$ |SET| and $\perp\in \mu$[DATA](d) for every d$\in$DATA,

- a function μ[OPNS] : OPNS $\rightarrow$ |FUN| and σ : d_1 x ... x d_n $\rightarrow$ d implies μ[OPNS](σ) : μ[DATA](d_1) x ... x μ[DATA](d_n) $\rightarrow \mu$[DATA](d) for every $\sigma \in$OPNS, and

- a function μ[PRED] : PRED $\rightarrow$ |REL| and π : d_1 x ... x d_n implies μ[PRED](π) $\subseteq$ μ[DATA](d_1) x ... x μ[DATA](d_n) for every $\pi \in$PRED.

The set OPNS^{2,1} denotes all operators σ, $\sigma \in$OPNS, having source(σ)=<d,d> and destination(σ) = d for some d$\in$DATA. Additionally the corresponding function μ[OPNS](σ) has to be commutative and associative. □

3.3 Remarks: Throughout the paper all sets mentioned in definitions have to be disjoint, except when common elements are explicitly allowed. Thus, e.g., DATA, OPNS, and PRED are disjoint as well as (the interpretations of) all data sorts with the exception of $\perp$, i.e., μ[DATA](d_1)$\cap\mu$[DATA](d_2)=$\{\perp\}$,if $d_1 \neq d_2$. Furthermore, throughout the paper the greek letter μ stands for meaning (and hopefully not for mysterious).

We have required every data sort d to contain the value $\perp\in \mu$[DATA](d), because it is useful to have an "undefined" value as result for incorrect applications of operations. Thus, we obtain for incorrect applications of operations this special value, for example μ[OPNS](/)(c,0) = $\perp$. In most cases it is useful to define the propagation of $\perp$ in the following way: An operation $\sigma \in$OPNS with σ:d_1 x ... x d_n $\rightarrow$d evaluates μ[OPNS](σ)(c_1,...,c_n) to $\perp$, if there is a c_i with c_i = $\perp$. For predicates $\pi \in$PRED with π : d_1 x ... x d_n (c_1,...,c_n) $\in \mu$[PRED](π) does not hold, if there is a c_i with c_i=$\perp$.

For the rest of the paper we assume one fixed data signature and algebra including (among others) the sorts int, rat (for integer and rational numbers) and string together with adequate operations and predicates to be given.

3.4 Example: Our example presented in chapter 2 uses the data sorts int, rat, string and point. As will be seen later in the context of aggregations like SUM or AVG, the addition + on integers and on rationals has to be in the set OPNS^{2,1}. We also need the division / on rationals.

3.5 Fact: The syntax of the data signature directly corresponds to a signature DS = (DATA, OPNS,PRED) in the sense of abstract data type theory. The semantics of a data signature is equivalent to an algebra μ[DS]=(μ[DATA], μ[OPNS], μ[PRED]) with carrier sets, functions and relations.

3.6 Definition: (sort expressions)

Let a data signature DS and a set S$\in$|SET| with DATA $\subseteq$ S together with a (semantic) function μ[S] : S $\rightarrow$ |SET| (such that μ[S](d) = μ[DATA](d) for d$\in$DATA and $\perp\in \mu$[S](s) for every s$\in$S) be given. The syntax of the sort expressions over S is given by the set SORT-EXPR(S) determined by the following rules.

(i) If s$\in$S, then s$\in$SORT-EXPR(S).

(ii) If s$\in$SORT-EXPR(S), then set(s)$\in$SORT-EXPR(S).

(iii)If s$\in$SORT-EXPR(S), then list(s)$\in$SORT-EXPR(S).

(iv) If s$\in$SORT-EXPR(S), then bag(s)$\in$SORT-EXPR(S).

(v) If $s_1,...,s_n$ $\in$SORT-EXPR(S), then prod($s_1,...,s_n$)$\in$SORT-EXPR(S).

The semantics of the sort expressions is a function μ[SORT-EXPR(S)]:SORT-EXPR(S) $\rightarrow$ |SET| determined by the following rules.

(i) μ[SORT-EXPR(S)](s) := μ[S](s)

(ii) μ[SORT-EXPR(S)](set(s)) := F(μ[SORT-EXPR(S)](s)) $\cup$ $\{\perp\}$

(iii)μ[SORT-EXPR(S)](list(s)) := (μ[SORT-EXPR(S)](s))$^*\cup$ $\{\perp\}$

(iv) μ[SORT-EXPR(S)](bag(s)) := B(μ[SORT-EXPR(S)](s)) $\cup$ $\{\perp\}$

(v) μ[SORT-EXPR(S)](prod($s._1,...,s_n$) :=

(μ[SORT-EXPR(S)](s_1) x ... x μ[SORT-EXPR(S)](s_n)) $\cup$ $\{\perp\}$

$\hat{\mu}$[SORT-EXPR(S)] is an abbreviation for $\cup_{s\in SORT-EXPR(S)}\mu$[SORT-EXPR(S)](s), i.e., the set of all instances belonging to the sort expression over the set S. $\square$

3.7 Example: In our example S is equal to the set {int, rat, point, string, TOWN, RIVER, COUNTRY, SEA, LAKE, WATERS, DISTRICT, STREET}. The diagrams use three multi-valued sort expression, namely list(point), set(DISTRICT) and set(STREET).

3.8 Fact: The set S together with the semantic function μ[S] induces a signature DS $\cup$ S = (S, OPNS, PRED) and an algebra μ[DS $\cup$ S] = (μ[S], μ[OPNS], μ[PRED]), which is a conservative and complete extension of μ[DS] (due to the fact that μ[DS](d) = μ[S](d) for d$\in$DATA).

3.9 Remark: The same semantics of sort expressions can also be specified algebraically, if we view a sort expression as a new sort and introduce generating operations in the following way.

$\text{EMPTY}_{set(s)}$: $\rightarrow$ set(s)

$\text{ADD}_{set(s)}$: set(s) x s $\rightarrow$ set(s)

$\text{EMPTY}_{bag(s)}$: $\rightarrow$ bag(s)

$\text{ADD}_{bag(s)}$: bag(s) x s $\rightarrow$ bag(s)

$\text{EMPTY}_{list(s)}$: $\rightarrow$ list(s)

$\text{ADD}_{list(s)}$: list(s) x s $\rightarrow$ list(s)

$\text{MAKE}_{prod(s_1,...,s_n)}$: s_1 x ... x s_n $\rightarrow$ prod($s_1,...,s_n$)

Additionally, error constants representing $\perp$ could be added. For sets, two equations must be valid, whereas the interpretation of bags is restricted by only one equation. Lists and products are generated "freely' (without restricting equations).

$\text{ADD}_{set(s)}(\text{ADD}_{set(s)}(\text{S,X}),\text{Y}) = \text{ADD}_{set(s)}(\text{ADD}_{set(s)}(\text{S,Y}),\text{X})$

$\text{ADD}_{set(s)}(\text{ADD}_{set(s)}(\text{S,X}),\text{X}) = \text{ADD}_{set(s)}(\text{S,X})$

$\text{ADD}_{bag(s)}(\text{ADD}_{bag(s)}(\text{S,X}),\text{Y}) = \text{ADD}_{bag(s)}(\text{ADD}_{bag(s)}(\text{S,Y}),\text{X})$

Because the second set equation is not valid for bags, bags and sets fulfill the following rules (with respect to the {{...}}- and {...}-notation).

$\{\{c_1\}\} \cup \{\{c_2\}\} = \{\{c_2\}\} \cup \{\{c_1\}\} = \{\{c_1,c_2\}\} = \{\{c_2,c_1\}\}$ if $c_1 \neq c_2$

$\{\{c\}\} \cup \{\{c\}\} = \{\{c,c\}\}$

$\{c_1\} \cup \{c_2\} = \{c_2\} \cup \{c_1\} = \{c_1,c_2\} = \{c_2,c_1\}$ if $c_1 \neq c_2$

$\{c\} \cup \{c\} = \{c\}$

3.10 Definition: (operations and predicates induced by the sort expressions)

Let the sort expressions as defined above be given. The syntax of the operations and predicates induced by the sort expressions is given by the following sets and functions (s, s_1, ..., s_n refer to arbitrary sort expressions and σ to an element of $OPNS^{2,1}$ with σ : d x d $\rightarrow$ d):

OPNS(S)/PRED(S)	source+destination/arguments	informal description
$CNT_{set(s)}$	set(s) $\rightarrow$ int	counts the elements
$IND_{set(s)}$	set(s) $\rightarrow$ set(int)	set of indices of a set
$APL_{\sigma,set(d)}$	set(d) $\rightarrow$ d	applies operation to a set
$IN_{set(s)}$	set(s) $\times$ s	element-of relation
$CNT_{list(s)}$	list(s) $\rightarrow$ int	counts the elements
$IND_{list(s)}$	list(s) $\rightarrow$ set(int)	set of indices of a list
$LTB_{list(s)}$	list(s) $\rightarrow$ bag(s)	converts a list to a bag
$SEL_{list(s)}$	list(s) $\times$ int $\rightarrow$ s	selects the i-th element
$POS_{list(s)}$	list(s) $\times$ s $\rightarrow$ set(int)	set of indices of an element
$APL_{\sigma,list(d)}$	list(d) $\rightarrow$ d	applies operation to a list
$IN_{list(s)}$	list(s) $\times$ s	element-of relation
$CNT_{bag(s)}$	bag(s) $\rightarrow$ int	counts the elements
$IND_{bag(s)}$	bag(s) $\rightarrow$ set(int)	set of indices of a bag
$BTS_{bag(s)}$	bag(s) $\rightarrow$ set(s)	converts a bag to a set
$OCC_{bag(s)}$	bag(s) $\times$ s $\rightarrow$ int	counts the occurrences
$APL_{\sigma,bag(d)}$	bag(d) $\rightarrow$ d	applies operation to a bag
$IN_{bag(s)}$	bag(s) $\times$ s	element-of relation
$PRJ_{record(s_1,...,s_n),i}$	record(s_1,...,s_n) $\rightarrow$ s_i (i $\in$ 1..n)	projection

Figure 1: Operations and predicates induced by sort expressions.

The semantics of the operations induced by the sort expressions is a function μ[OPNS(S)] : OPNS(S) $\rightarrow$ |FUN| and a function μ[PRED(S)] : PRED(S) $\rightarrow$ |REL| determined by the following lines. Domain and codomain of semantic functions and predicates are not given explicitly, but are determined by the following rules : If σ : $s_1 \rightarrow s_2$, then μ[OPNS(S)](σ) : μ[SORT-EXPR(S)](s_1) $\rightarrow$ μ[SORT-EXPR(S)](s_2), and if π : s, then μ[PRED(S)](π) $\subseteq$ μ[SORT-EXPR(S)](s). Furthermore we abbreviate μ[OPNS(S)](σ) by $\mu(\sigma)$ and μ[PRED(S)](π) by $\mu(\pi)$. All these functions preserve the undefined value $\perp$ ($\mu(\sigma)(\perp) = \perp$ and $\mu(\sigma)(c_1,c_2) = \perp$, if $c_1 = \perp$ or $c_2 = \perp$) and the predicates do not hold for $\perp$ (not $\perp \in \mu(\pi)$).

- **Set operations**

  $\mu(CNT_{set(s)})$: $\{c_1,...,c_n\} \mapsto n$

  $\mu(IND_{set(s)})$: $\{c_1,...,c_n\} \mapsto \{1,...,n\}$

  $\mu(APL_{\sigma,set(d)})$: $\{c_1,...,c_n\} \mapsto$

  $$\begin{cases} \perp & \text{if } n = 0 \\ c_1 & \text{if } n = 1 \\ \mu(\sigma)(c_1, \mu(APL_{\sigma,set(d)})(\{c_2, ..., c_n\})) & \text{if } n \geq 2 \end{cases}$$

  $\mu(IN_{set(s)}) := \{ (\{c_1,...,c_n\} , c) \mid c \in \{c_1,...,c_n\} \}$

- **List operations**

  $\mu(CNT_{list(s)})$: $<c_1,...,c_n> \mapsto n$

  $\mu(IND_{list(s)})$: $<c_1,...,c_n> \mapsto \{1,...,n\}$

  $\mu(LTB_{list(s)})$: $<c_1,...,c_n> \mapsto \{\{c_1,...,c_n\}\}$

$\mu(\mathrm{SEL}_{list(s)})$: (<c_1,...,c_n> , i) $\mapsto \begin{cases} c_i & \text{if } 1 \leq i \leq n \\ \perp & \text{otherwise} \end{cases}$

$\mu(\mathrm{POS}_{list(s)})$: (<c_1,...,c_n>,c) $\mapsto$

$$\begin{cases} \emptyset & \text{if } n = 0 \\ \mu(\mathrm{POS}_{list(s)})(< c_1, ..., c_{n-1} >, c) \cup \{n\} & \text{if } n \geq 1 \text{ and } c_n = c \\ \mu(\mathrm{POS}_{list(s)})(< c_1, ..., c_{n-1} >, c) & \text{if } n \geq 1 \text{ and } c_n \neq c \end{cases}$$

$\mu(\mathrm{APL}_{\sigma,list(d)})$: <c_1,...,c_n> $\mapsto$

$$\begin{cases} \perp & \text{if } n = 0 \\ c_1 & \text{if } n = 1 \\ \mu(\sigma)(c_1, \mu(\mathrm{APL}_{\sigma,list(d)})(< c_2, ..., c_n >)) & \text{if } n \geq 2 \end{cases}$$

$\mu(\mathrm{IN}_{list(s)})$:= { (<c_1,...,c_n> , c) | c $\in$ {c_1,...,c_n} }

- **Bag operations**

  $\mu(\mathrm{CNT}_{bag(s)})$: {{c_1,...,c_n}} $\mapsto$ n

  $\mu(\mathrm{IND}_{bag(s)})$: {{c_1,...,c_n}} $\mapsto$ {1,...,n}

  $\mu(\mathrm{BTS}_{bag(s)})$: {{c_1,...,c_n}} $\mapsto$

$$\begin{cases} \emptyset & \text{if } n = 0 \\ \{c_1\} \cup \mu(\mathrm{BTS}_{bag(s)})(\{\{c_2, ..., c_n\}\}) & \text{if } n \geq 1 \end{cases}$$

  $\mu(\mathrm{OCC}_{bag(s)})$: ({{c_1,...,c_n}} , c) $\mapsto$

$$\begin{cases} 0 & \text{if } n = 0 \\ \mu(\mathrm{OCC}_{bag(s)})(\{\{c_2, ..., c_n\}\}, c) + 1 & \text{if } n \geq 1 \text{ and } c_1 = c \\ \mu(\mathrm{OCC}_{bag(s)})(\{\{c_2, ..., c_n\}\}, c) & \text{if } n \geq 1 \text{ and } c_1 \neq c \end{cases}$$

  $\mu(\mathrm{APL}_{\sigma,bag(d)})$: {{c_1,...,c_n}} $\mapsto$

$$\begin{cases} \perp & \text{if } n = 0 \\ c_1 & \text{if } n = 1 \\ \mu(\sigma)(c_1, \mu(\mathrm{APL}_{\sigma,bag(d)})(\{c_2, ..., c_n\})) & \text{if } n \geq 2 \end{cases}$$

  $\mu(\mathrm{IN}_{bag(s)})$:= { ({{c_1,...,c_n}} , c) | c $\in$ {c_1,...,c_n} }

- **Record operations**

  $\mu(\mathrm{PRJ}_{record(s_1,...,s_n),i})$: (c_1,...,c_n) $\mapsto$ c_i

$\mu[\mathrm{APL}_{\sigma,list(s)}]$ and $\mu[\mathrm{APL}_{\sigma,bag(s)}]$ are defined just as $\mu[\mathrm{APL}_{\sigma,set(s)}]$ (analogously for the other functions not mentioned explicitly). □

3.11 Remarks: {...} $\cup$ {...} refers to the union of sets (this guarantees the elemination of duplicates, while {{...}} $\cup$ {{...}} is the union of bags respecting duplicates). For all above functions equational specifications can be given.

We use the following conventions and abbreviations for frequently used operations. If no ambiguities occur, we drop the list-, set-, bag- or prod-indices of the operation symbols and leave out parenthesis in accordance with the usual rules. SUM refers to APL_+ dependent on the context (the idea of applying a binary operator to a list of values has also been proposed in [Bi 87]). MAX means APL_{max} with max(x,y) = if x>y then x else y. MIN is defined analogously. $\mathrm{AVG}_{set(d)}$(x) refers to $\mathrm{APL}_{+,set(d)}$(x) / $\mathrm{CNT}_{set(d)}$(x), analogously for lists and bags. With this convention we have of course AVG($\emptyset$) = $\perp$. Instead of PRJ_i(x) or PRJ_i((x_1,...,x_n)) we also use the more suggestive x.i or (x_1,...,x_n).i, where i is a constant between 1 and n. LTS stands for BTS o LTB.

3.12 Example: In the first query of the example presented in chapter 2 AVG -[...]- stands for $APL_{+,bag(int)}$-[...]- / $CNT_{bag(int)}$-[...]-. In the second query SUM refers to $APL_{+,bag(int)}$ and BTS was short for $BTS_{bag(string)}$

3.13 Fact: The syntax of the sort expressions introduces a signature DS ∪ SORT-EXPR(S) = (SORT-EXPR(S), OPNS ∪ OPNS(S), PRED ∪ PRED(S)) and the semantics corresponds to an algebra μ[DS ∪ SORT-EXPR(S)] = (μ[SORT-EXPR(S)], μ[OPNS ∪ OPNS(S)],μ[PRED ∪ PRED(S)]).

Again, μ[DS ∪ SORT-EXPR(S)] is a conservative and complete extension of μ[DS ∪ S]. Of course, there is an infinite number of sorts and operations in this algebra.

3.14 Definition: (extended entity-relationship schema)

Let a data signature DS be given. The syntax of an extended entity-relationship schema EER(DS) over DS is given by

- the sets ENTITY, RELATION, ATTRIBUTE, COMPONENT, CONSTRUCTION ∈ |FISET| and

- the functions participants, asource, adestination, csource, cdestination, input, output such that

participants : RELATION → $ENTITY^+$,

asource: ATTRIBUTE→ENTITY∪RELATION,

adestination:ATTRIBUTE→ {d, set(d), list(d), bag(d) | d∈DATA},

csource: COMPONENT→ENTITY,

cdestination : COMPONENT→ {e, set(e), list(e), bag(e) | e∈ENTITY},

input: CONSTRUCTION → F(ENTITY), and

output : CONSTRUCTION → F(ENTITY).

If r∈RELATION with participants(r)=<e_1,...,e_n >, this is notated as r(e_1,...,e_n). If a∈ATTRIBUTE with asource(a)=e or asource(a)=r and adestination(a)=d, this is notated as a: e →d or a: r →d, respectively. If c∈COMPONENT with csource(c)=e and cdestination(c)=e', this is notated as c:e →e'. If c∈CONSTRUCTION with input(c)={i_1,...,i_n } and output(c)= {o_1,...,o_m}, this is notated as c(i_1,...,i_n; o_1,...,o_m).

For two distinct constructions c_1,c_2 ∈CONSTRUCTION the following conditions must hold:

(i) output(c_1) ∩ output(c_2) = ∅

(ii) It is not allowed that $connection^+$(e,e) holds for some e∈ENTITY, where $connection^+$ is the transitive closure of the relation connection defined by: if e_{in} ∈input(c) and e_{out} ∈output(c) for some c∈CONSTRUCTION, then connection(e_{in},e_{out}) holds.

The semantics of an extended entity-relationship schema EER(DS)is given by

- a function μ[ENTITY]:ENTITY → |FISET| such that ⊥∈ μ[ENTITY](e) for e∈ENTITY,

- a function μ[RELATION] : RELATION → |FISET| such that r(e_1, ..., e_n) implies μ[RELATION](r) ⊆ (μ[ENTITY](e_1)-{⊥}x ...x μ[ENTITY](e_n)-{⊥}) ∪{⊥}, and ⊥∈ μ[RELATION](r) for r∈RELATION,

- a function μ[ATTRIBUTE]:ATTRIBUTE → |FUN| such that

a:e→ d implies μ[ATTRIBUTE](a): μ[ENTITY](e) → μ[SORT-EXPR(DATA)](d), and

a:r→d implies μ[ATTRIBUTE](a):μ[RELATION](r)→ μ[SORT-EXPR(DA TA)](d),

- a function μ[COMPONENT]:COMPONENT → |FUN| such that c:e$\rightarrow$e' implies μ[COMPONENT](c):μ[ENTITY](e)$\rightarrow$ μ[SORT-EXPR(ENTI TY)](e'), and

- a function μ[CONSTRUCTION]:CONSTRUCTION $\rightarrow$ |FUN| such that c(i_1,...,i_n; o_1,...,o_m) implies μ[CONSTRUCTION](c): $\cup_{j=1}^{m}\mu$[ENTITY](o_j)$\rightarrow$ $\cup_{k=1}^{n}\mu$[ENTITY](i_k), where each μ[CONSTRUCTION](c) is injective.

Every function μ[ATTRIBUTE](a), μ[COMPONENT](c), and μ[CONSTRUCTION](c) has to preserve the undefined value, e.g., μ[ATTRIBUTE](a)($\perp$) = $\perp$ and so on. $\square$

3.15 Remarks: The elements of ENTITY are called entity (or object) types, the elements of RELATION are the relationship types, and the elements of ATTRIBUTE are the attributes names, all known from the ER Model defined in [Ch 76]. μ[ENTITY](e) is the set of entities belonging to the entity type e, μ[RELATION](r) defines which entities are related by r, and μ[ATTRIBUTE](a) gives the attributes of entities or relationships (related entities).

Additionally, we have a set COMPONENT of component names to model complex entity types. μ[COMPONENT](c) gives the components of entities, i.e., an entity may have as part (component) of itself another entity, a set, list, or bag of entities.

To provide modelling primitives for specialization and generalization [SS 77] we introduce the set CONSTRUCTION. Its elements are called type constructions. A type construction c may be regarded as a rearrangement of entity types. Starting with non-constructed or already defined entity types in input(c) the new entity types in output(c) are constructed. Although all introduced entity types are disjoint by definition, the constructed entity types may be considered as a new classification of the entities in the input types. Formally, we express this fact by the function

μ[CONSTRUCTION](c) : $\cup_{j=1}^{m}\mu$[ENTITY](o_j) $\rightarrow$ $\cup_{k=1}^{n}\mu$[ENTITY](i_k)

yielding for an output entity the corresponding input entity it refers to. Since this function is injective, every output entity corresponds to exactly one input entity. But an input entity need not appear in any output type at all, because the function is not required to be surjective. This semantics of CONSTRUCTION is equivalent to demanding that every output entity "is" an input entity:

$\cup_{j=1}^{m}\mu$[ENTITY](o_j) $\subseteq$ $\cup_{k=1}^{n}\mu$[ENTITY](i_k)

The restricting conditions for constructions guarantee that we have for a constructed entity e uniquely determined constructions c_1,...,c_n, other constructed entities e_1,...,e_{n-1} and a non-constructed (basic) entity e_n, such that these items are related as given in Figure 2.

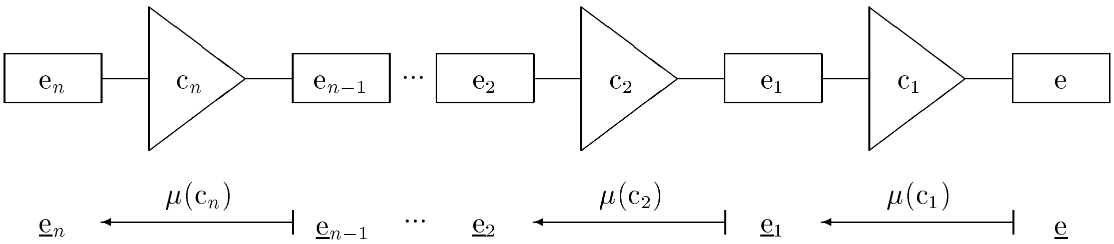


Figure 2: Uniquely determined construction path for constructed entities.

In other words, for a constructed entity there is a uniquely determined construction rule.

As for DATA, we assume the element $\perp$ to be element of every entity type in ENTITY. Thus, every attribute and component is per default optional. Indeed, this can be excluded by additional integrity constraints.

3.16 Example: In our example the following identities hold : ENTITY = {TOWN, RIVER, COUNTRY, SEA, LAKE, WATERS, DISTRICT, STREET}, RELATION = {lies-at, lies-in, flows- through, flows-into}, COMPONENT = {districts, streets} and CONSTRUCTION = {are}. The corresponding function participants, etc. can be derived from the diagrams.

3.17 Fact: The algebraization of an extended entity-relationship schema will be done in two steps. First we introduce the signature

EER-BASE = (SORT-EXPR(DATA) ∪ ENTITY ∪ RELATION, OPNS ∪ OPNS(DATA) ∪ OPNS(EER-BASE), PRED ∪ PRED(DATA)) with

OPNS(EER-BASE) = { mk_r : e_1 x ... x e_n → r | r(e_1,...,e_n) ∈ EER(DS) }∪

{ a : e → d ∈ EER(DS), a : r → q d ∈ EER(DS) }∪

{ $c_{o_j;i_k}$: o_j → i_k| c(i_1,...,i_n;o_1,...,o_m) ∈ EER(DS), j∈1..m, k∈1..n }

Given the semantics of an EER schema, an algebra μ[EER-BASE] can be defined as follows. The carriers are given by μ[SORT-EXPR(DATA)] (which is based on μ[DATA]), μ[ENTITY] and μ[RELATION]. The functions μ[OPNS] and μ[OPNS(DATA)] and the predicates μ[PRED] and μ[PRED(DATA)] are fixed by the algebra μ[DS ∪ SORT-EXPR(DATA)]. It remains to define the functions μ[OPNS(EER-BASE)] :

- $\mu(\mathrm{mk}_r)$: (e_1,...,e_n) → if (e_1,...,e_n) ∈ μ(r) then(e_1,...,e_n) else ⊥

- μ[OPNS(EER-BASE)](a) is given by μ[ATTRIBUTE](a).

- $\mu[c_{o_j;i_k}]$: o_j → if μ[CONSTRUCTION](c)(o_j) ∈ $\mu[i_k]$ then μ[CONSTRUCTION](c)(o_j) else ⊥

In the second step we define the final signature by EER = (SORT-EXPR(DATA ∪ ENTITY) ∪ RELATION, OPNS ∪ OPNS(DATA ∪ ENTITY) ∪ OPNS(EER-BASE) ∪ OPNS(EER), PRED ∪ PRED(DATA ∪ ENTITY)) with OPNS(EER) = { c : e → e' ∈ EER(DS) | c ∈ COMPONENT }.

The corresponding algebra μ[EER] has carriers equal to μ[EER-BASE] except that the additional carriers are determined by μ[SORT-EXPR(DATA ∪ ENTITY)] (which is based on μ[DATA] of the DS level and μ[ENTITY] of the EER-BASE level). The additional functions in OPNS(EER) are determined by μ[COMPONENT](c).

Again, the considered algebras are strongly related : μ[EER] is a conservative and complete extension of μ[EER-BASE] which extends μ[DS ∪ SORT-EXPR(DATA)].

3.18 Example: In the signature EER for our example we will have (among others) the following sorts and functions.

name : TOWN → string

population : TOWN → int

geometry : TOWN → list(point)

$\mathrm{mk}_{flows-through}$: RIVER x COUNTRY → flows-through

length : flows-through → rat

districts : TOWN → set(DISTRICT)

$\mathrm{are}_{WATERS;RIVER}$: WATERS → RIVER

$\mathrm{are}_{WATERS;SEA}$: WATERS → SEA

4. The Entity-Relationship Calculus

4.1 Notations: Before discussing the EER calculus in more detail, we introduce some abbreviations. Let an EER schema EER(DS) be given. SORT refers to the union of all data, entity, and relationship sorts : SORT := DATA ∪ ENTITY ∪ RELATION. Furthermore, we extend the notions of operations (OPNS) and predicates (PRED) introduced for data signatures to sort expressions and EER schemas resulting in the following notations:

$OPNS_{EXPR}$:= {CNT, LTB, BTS, LTS, SEL, IND, POS, OCC, PRJ_i, AVG, SUM, MIN, MAX} $\cup$ {APL_σ | σ ∈$OPNS^{2,1}$}

$PRED_{EXPR}$:= {IN}

$OPNS_{EER}$:= ATTRIBUTE ∪ COMPONENT

$PRED_{EER}$:= RELATION

Please note that (for example) CNT in the above definition of $OPNS_{EXPR}$ is short for $CNT_{list(s)}$, $CNT_{set(s)}$, $CNT_{bag(s)}$ with s∈SORT-EXPR(SORT). We can drop the indices, because no ambiguities shall occur. All operations and predicates are now concentrated to

OPNS := $OPNS_{DS}$∪ $OPNS_{EER}$∪ $OPNS_{EXPR}$

PRED := $PRED_{DS}$∪ $PRED_{EER}$∪ $PRED_{EXPR}$

μ[SORT], μ[$OPNS_{EXPR}$], μ[$PRED_{EXPR}$]μ[$OPNS_{EER}$], μ[$PRED_{EER}$], μ[OPNS], and μ[PRED], are determined by the corresponding μ's on the right hand side.

Let us remind you that the (interpretation of the) sets DATA, ENTITY, and RELATION are disjoint as well as all (interpretations of) set-/list-/bag-constructed sort expressions (except the special value ⊥). Due to the disjointness there is a unique function

sort : $\hat{\mu}$[SORT-EXPR(SORT)]-{⊥} → SORT-EXPR(SORT)

yielding for an instance of $\hat{\mu}$[SORT-EXPR(SORT)] the sort it belongs to; the value ⊥ belongs to every sort. However, we shall only build sort expressions over DATA∪ENTITY. Thus, we use the function sort only in the restricted form

sort : $\hat{\mu}$[SORTEXPRS]-{⊥} → SORTEXPRS

with SORTEXPRS := SORT-EXPR(DATA∪ENTITY)∪RELATION.

4.2 Definition: (variables and assignments)

Let a set VAR of variables and a function type : VAR → SORTEXPRS be given. The set of assignments ASSIGN is defined by ASSIGN := {α ∈ |FUN| α : VAR → $\hat{\mu}$[SORTEXPRS] and α(v)≠⊥ implies type(v)=sort(α(v)) }. The special assignment ε : VAR → {⊥} is called the empty assignment. In the following v∈VAR_s stands for v∈VAR and type(v)=s. □

4.3 Remark: Our calculus has to leave the usual hierarchical structure of calculi (terms - atomic formulas - formulas), because we will allow arbitrary bag-valued terms of the form -[t_1, ..., t_n | $\delta_1\wedge$ ... $\wedge\delta_n \wedge \varphi$]-. The terms t_1, ..., t_n compute the target information, $\delta_1\wedge$ ... $\wedge\delta_n$ are declarations of variables which can again use bag-valued terms of the above form to restrict the domain of a variable to a finite set (like this has been done in the second query in chapter 2).

4.4 Definition: (ranges)

The syntax of ranges is given by a set RANGE and functions domain : RANGE → SORTEXPRS and free : RANGE → F(VAR) (ρ ∈$RANGE_s$ stands for ρ ∈RANGE and domain(ρ)=s).

(i) If s∈ENTITY or s∈RELATION, then s∈$RANGE_s$ and $free_{RANGE}$(s):= ∅.

(ii) If t∈$TERM_{set(s)}$ with s∈SORTEXPRS, then t∈$RANGE_s$ and $free_{RANGE}$(t):=$free_{TERM}$(t).

The semantics of ranges is a function μ[RANGE] : RANGE x ASSIGN → μ[SORTEXPRS].

(i) μ[RANGE](s,α):= μ[SORT](s).

(ii) μ[RANGE](t,α):= μ[TERM](t,α). □

4.5 Example: The queries in chapter 2 used the following ranges: TOWN, COUNTRY and

BTS -[government(c) | c : COUNTRY]-

4.6 Definition: (declarations)

The syntax of declarations is given by a set DECL and functions free, decl : DECL $\rightarrow$ F(VAR).

(i) If v$\in$VAR, ρ_1,...,ρ_n $\in$RANGE$_{type(v)}$, and not v$\in$free(ρ_1)$\cup$...$\cup$free(ρ_n), then v: $\rho_1\cup...\cup\rho_n$ $\in$DECL, free(v: $\rho_1\cup...\cup\rho_n$):=free(ρ_1)$\cup$...$\cup$free(ρ_n), and decl(v: $\rho_1\cup...\cup\rho_n$):= {v}.

(ii) If v$\in$VAR, ρ_1,...,ρ_n $\in$RANGE$_{type(v)}$, δ $\in$DECL with free(ρ_1)$\cup$...$\cup$free(ρ_n)$\subseteq$decl(δ), and not v$\in$free(δ), then v: $\rho_1\cup...\cup\rho_n$;δ $\in$DECL, free(v: $\rho_1\cup...\cup\rho_n$;δ):=free(δ), and decl(v: $\rho_1\cup...\cup\rho_n$;δ):= {v}$\cup$decl(δ).

The semantics of declarations is a relation μ[DECL] $\subseteq$ DECL x ASSIGN.

(i) (v: $\rho_1\cup...\cup\rho_n$,α)$\in$ μ[DECL] iff α(v)$\in$ μ[RANGE](ρ_1,α) or ... or α(v)$\in$ μ[RANGE](ρ_n,α).

(ii) (v: $\rho_1\cup...\cup\rho_n$;δ,α)$\in$ μ[DECL] iff (α(v)$\in$ μ[RANGE](ρ_1,α) or ... or α(v)$\in$ μ[RANGE](ρ_n,α)) and (δ,α)$\in$ μ[DECL]. $\square$

4.7 Remarks: At first, we do not only allow the form (v:ρ) but also the more general (i) (v:$\rho_1\cup$ ... $\cup\rho_n$). This form is necessary to express every term of the relational algebra [Ma 83], especially the union of sets, in our calculus. In declarations of the form (ii) (v$_1$:ρ_1); ...; (v$_n$:ρ_n) the variables v$_1$,...,v$_n$ (in front of the colons) are declared (each (v$_i$:ρ_i) may be a union of form (i)). Each range ρ_i ($i = 1..n-1$) may contain free variables, but only from the set of previously declared variables {v$_{i+1}$,...,v$_n$}, except ρ_n, which can have other free variables. Indeed, the later ones are the free variables of the declaration, which must be disjoint from the variables declared in the declaration.

Each variable v is bound to a finite set of values determined in the following way.

v$_n$ can take each value from μ[RANGE](ρ_n,α),

v$_{n-1}$ can take each value from μ[RANGE](ρ_{n-1},α) “possibly dependent on the value assigned to v$_n$ by α”, and so on.

4.8 Example: The queries of chapter 2 declared the following variables:

t : TOWN ; c : COUNTRY ; g : BTS -[government(c) | c : COUNTRY]-

Declarations with free variable t of type TOWN are the following ones:

- “p : LTS(geometry(t))” declares the variable p (of type point).

- “s : streets(d) ; d : districts(t)” declares s (of type STREET) and d (of type DISTRICT).

- “n : BTS -[name(r’) | r’ : RIVER $\wedge$ r : RIVER $\wedge$ lies-at(t,r) $\wedge$ flows-into(r’,WATERS(r))]- $\cup$ BTS -[name(r) | r : RIVER $\wedge$ lies-at(t,r)]-” declares n (of type string).

4.9 Definition: (terms)

The syntax of terms is given by a set TERM, and functions sort : TERM $\rightarrow$ SORTEXPRS and free : TERM $\rightarrow$ F(VAR) (t$\in$TERM$_s$ stands for t$\in$TERM and sort(t)=s).

(i) If v$\in$VAR$_s$, then v$\in$TERM$_s$ and free(v):= {v}.

(ii) If v$\in$VAR$_r$ and r(e$_1$,...,e$_n$)$\in$RELATION, then v.i$\in$TERM$_{e_i}$ (i=1..n) and free(v.i):= {v}.

(iii) If c$\in$CONSTRUCTION, s$_{in}$ $\in$ input(c), s$_{out}$ $\in$ output(c), t$_{in}$ $\in$ TERM$_{s_{in}}$, and t$_{out}$ $\in$ TERM$_{s_{out}}$, then s$_{out}$(t$_{in}$)$\in$TERM$_{s_{out}}$ with free(s$_{out}$(t$_{in}$)):=free(t$_{in}$) and s$_{in}$(t$_{out}$)$\in$TERM$_{s_{in}}$ with free(s$_{in}$(t$_{out}$)):=free(t$_{out}$).

(iv) If ω :s$_1$x...xs$_n$ $\rightarrow$s$\in$OPNS and t$_i$ $\in$TERM$_{s_i}$ (i=1..n), then ω(t$_1$,...,t$_n$)$\in$TERM$_s$ with free(ω(t$_1$,...,t$_n$)):=free(t$_1$)$\cup$...$\cup$free(t$_n$).

(v) If t$_i$ $\in$TERM$_{s_i}$ (i=1..n), δ_j $\in$DECL (j=1..k), φ $\in$FORM, and decl(δ_i) $\cap$ decl(δ_j) = decl(δ_i)

$\cap$ free(δ_j)= $\emptyset$ (i$\neq$j), then -[t_1,....,t_n — $\delta_1\wedge$ ... $\wedge\delta_k \wedge \varphi$]- $\in$ TERM$_{bag(prod(s_1,...,s_n))}$ and free(-[t_1,...,t_n | $\delta_1\wedge$ ... $\wedge\delta_k \wedge \varphi$]-) :=(free($\delta_1$)$\cup$...$\cup$free($\delta_k$) $\cup$ free(t_1)$\cup$...$\cup$free(t_n) $\cup$ free(φ)) - (decl(δ_1)$\cup$...$\cup$decl(δ_k)).

The semantics of terms is a function μ[TERM] : TERM x ASSIGN $\rightarrow$ $\hat{\mu}$[SORTEXPRS].

(i) μ[TERM](v,α):= α(v).

(ii) μ[TERM](v.i,α):=if μ[TERM](v,α)=(x_1,...,x_n) then x_i else $\perp$.

(iii) μ[TERM](s_{out}(t_{in}),α):=if there is e$\in$ μ[ENTITY](s_{out}) : μ[CONSTRUCTION](c)(e)= μ[TERM](t_{in},α) then e else $\perp$.

μ[TERM](s_{in}(t_{out}),α):=if μ[CONSTRUCTION](c)(μ[TERM](t_{out},α))$\in$ μ[ENTITY](s_{in}) then μ[CONSTRUCTION](c)(μ[TERM](t_{out},α)) else $\perp$.

(v) μ[TERM](ω(t_1,...,t_n),α):= μ[OPNS](ω)(μ[TERM](t_1,α),...,μ[TERM](t_n,α)).

(vi) μ[TERM](-[t_1,...,t_n | $\delta_1\wedge$ ... $\wedge\delta_k \wedge \varphi$]-, α) := {{ (μ[TERM](t_1,α'),...,μ[TERM](t_n,α')) | there is α'$\in$ASSIGN with α'(v)= α(v) for v$\in$VAR-(decl(δ_1)$\cup$...$\cup$decl(δ_k)) and (δ_1,α')$\in$ μ[DECL] and ... and (δ_k,α')$\in$ μ[DECL] and (φ,α')$\in$ μ[FORM]}}. □

4.10 Remark: If no ambiguity arises, we use entity types instead of integers in the case of terms according to point (ii), e.g., ft.RIVER instead of ft.1 (with type(ft)=flows-through).

4.11 Example: The queries in chapter 2 are term according to point (vi) of the above definition and have sort bag(prod(string,rat)) and bag(prod(string,int)), respectively.

- -[population(c) | c : COUNTRY $\wedge$ government(c)=g]- is a terms of sort bag(int) with free variable g. It was used in the second query in chapter 2.

- -[r | r : RIVER $\wedge$ lies-in(r,c)]- is a term of sort bag(RIVER) with free variable c.

4.12 Definition: (formulas)

The syntax of formulas is given by a set FORM and a function free : FORM $\rightarrow$ F(VAR).

(i) If π :s_1x...xs_n $\in$PRED and t_i $\in$TERM$_{s_i}$ (i=1..n), then π(t_1,...,t_n)$\in$FORM and free(π(t_1,...,t_n)):=free(t_1)$\cup$...$\cup$free(t_n).

(ii) If t_1,t_2 $\in$TERM$_s$, then t_1 =t_2 $\in$FORM and free(t_1 =t_2):=free(t_1)$\cup$free(t_2).

(iii) If t$\in$TERM, then UNDEF(t)$\in$FORM and free(UNDEF(t)):=free(t).

(iv) If φ $\in$FORM, then $\neg$(φ)$\in$FORM and free($\neg$(φ)):=free(φ).

(v) If φ_1,φ_2 $\in$FORM, then ($\varphi_1 \vee \varphi_2$)$\in$FORM and free(($\varphi_1 \vee \varphi_2$)):=free(φ_1)$\cup$free(φ_2).

(vi) If φ $\in$FORM and δ $\in$DECL, then $\exists\delta$(φ)$\in$FORM and free($\exists\delta$(φ)):=(free(φ)-decl(δ))$\cup$free(δ).

The semantics of formulas is a relation μ[FORM] $\subseteq$ FORM x ASSIGN.

(i) (π(t_1,...,t_n),α)$\in$ μ[FORM] iff (μ[TERM](t_1,α),...,μ[TERM](t_n,α))$\in$ μ[PRED](π).

(ii) (t_1 =t_2,α)$\in$ μ[FORM] iff μ[TERM](t_1,α)= μ[TERM](t_2,α).

(iii) (UNDEF(t),α)$\in$ μ[FORM] iff μ[TERM](t,α)=$\perp$.

(iv) ($\neg$(φ),α)$\in$ μ[FORM] iff not (φ,α)$\in$ μ[FORM].

(v) (($\varphi_1 \vee \varphi_2$),α)$\in$ μ[FORM] iff (φ_1,α)$\in$ μ[FORM] or (φ_2,α)$\in$ μ[FORM].

(vi) ($\exists\delta$(φ),α)$\in$ μ[FORM] iff there is α'$\in$ASSIGN with α'(v)= α(v) for v$\in$VAR-decl(δ) and (φ,α')$\in$ μ[FORM] and (δ,α')$\in$ μ[DECL]. □

4.13 Remark: We also use the other logical connectives $\wedge$, $\Rightarrow$, xor, etc. and the quantifier $\forall$ with the usual semantics. They can be defined by means of the above definitions.

4.14 Definition: (queries)

The syntax of queries is the set QUERY:

- If t$\in$TERM with sort(t)$\in$SORT-EXPR(DATA) and free(t)=$\emptyset$, then t$\in$QUERY.

The semantics of queries is a function μ[QUERY] : QUERY $\rightarrow$ μ[SORT-EXPR(DATA)].

- μ[QUERY](t) := μ[TERM](t,ε). □

4.15 Facts:

- Every multi-valued term t$\in$TERM is evaluated to a finite set, bag, or list for every assignment α.

- Since a query is a special case of a term, any query yields a finite result: The EER calculus is safe.

- The EER calculus is relationally complete (if every relation is modelled by an entity type).

The proofs can be found in [HG 88]. Thus our calculus preserves nice properties of the relational calculi [Ma 83], but on the other hand it is also more expressive. In the classical calculi it is not possible to compute for instance the cardinality (i.e., the number of acually stored tuples) of a relation R(a_1 : D_1, ..., a_n : D_n). Of course, this presents no problem in our approach, because we can use data type operations : $CNT_{bag(R)}$ -[r | r : R]-. We here assume the relation R is modelled by an entity type R with attributes a_1,...,a_n.

5. Integrity Constraints

5.1 Concept: We now use our calculus to formulate integrity constraints. An integrity constraint is a formula of the calculus without free variables. These formulas restrict the class of EER-algebras to algebras where the formulas hold.

5.2 Definition: (functional restriction)

The syntax of a functional restriction for a relation r(e_1,...,e_n) is a pair (f= f_1,...,f_m, g= g_1,...,g_k) of subsets of e_1,...,e_n. The semantics of a functional restriction is the following formula.

($\forall$ v , w : r) v.f_1 = w.$f_1$$\wedge$ ... $\wedge$ v.f_m = w.f_m $\Rightarrow$ v.g_1 = w.$g_1$$\wedge$ ... $\wedge$ v.g_k = w.g_k□

5.3 Remark: Please notice, it is possible to formulate more than one functional restriction for a single relationship. For instance, if we have a relationship r(A,B,C) with three entity types, one can demand that r is a function from A to C as well as from C x B to A.

5.4 Example: In the example in chapter 2 the functional restriction ({RIVER},{WATERS}) on the relationship flows-into was given. It also makes sense to require the functional restriction ({TOWN},{COUNTRY}) for lies-in in order to express that a town lies in exactly one country.

5.5 Definition: (key specification)

The syntax of a key specification for an entity type e is given by a subset a_1,..,a_n of the attributes of e, a subset c_1,...,c_m of the components of e and a subset r_1,...,r_m of the relationships of e. The semantics of a key specification is given by the formula

($\forall$ v , w : e) v $\neq$ w $\Rightarrow$

[a_1(v) $\neq$ a_1(w) $\vee$ ... $\vee$ a_n(v) $\neq$ a_n(w) $\vee$

c_1(v) $\neq$ c_1(w) $\vee$ ... $\vee$ c_m(v) $\neq$ c_m(w) $\vee$

($\exists$ $*_1$) r_1(v,$*_1$) xor r_1(w,$*_1$) $\vee$ ... $\vee$ ($\exists$ $*_m$) r_m(v,$*_m$) xor r_m(w,$*_m$)] □

5.6 Remarks: The*-notation has to be explained shortly: Suppose r(A,B,C) is given. Then the relationship r is part of a key for the entity type B, if the following formula holds.

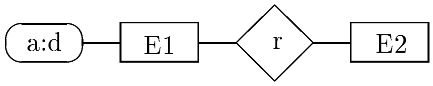
$(\forall$ v , w : B) ...

($\exists$ x_A : A) ($\exists$ x_C : C) r(x_A,v,x_C) xor r(x_A,w,x_C)

...

Please notice, the other direction of the implication in the key specification formula trivially holds. Thus, key specifications can be regarded as characterizations of equality.

Key relations also allow more subtle objects than key functions [EDG 86]. Consider the following simple example:



Suppose the data type d has only two values 0 and 1, the attribute a is the only key for E1 and the relationship r the only one for E2. Because the attribute a can only take two values there are (up to isomorphism) only two possible entities $e1_1$ and $e1_2$ of type E1, which can beobserved to be inequivalent : a($e1_1$)=0 and a($e1_2$)=1. In the case of a key function r from E2 to E1 there are also only two observable inequivalent entities for E2. But, if r is an arbitrary relation, there are (up to isomorphism) four possible entities for E2 : $e2_1$ related with no E1-entity, $e2_2$ related with $e1_1$, $e2_3$ related with $e1_2$ and $e2_4$ related with $e1_1$ and $e1_2$. It is an open problem how to construct "universes" [EDG 86, SSE 87] for key relations. The above example suggests some kind of power set construction yielding a final algebra analogously to [EDG 86, SSE 87].

5.7 Example: The example in chapter 2 required that {name,flows-through} is a key for RIVER. For the entity type TOWN on can demand {name,lies-in} as well as {geometry} to be a key:

($\forall$ t , t' : TOWN)

t $\neq$ t' $\Rightarrow$ name(t) $\neq$ name(t') $\vee$ ($\exists$ c : COUNTRY) lies-in(t,c) xor lies-in(t',c)

$\wedge$

t $\neq$ t' $\Rightarrow$ geometry(t) $\neq$ geometry(t')

The last line says : If two towns have the same geometry (the same list of points representing the town's border), then the towns are identical. This example shows that more than one key for a single entity type can make sense.

5.8 Definition: (cardinality constraint) The syntax of a cardinality constraint for a set-, list- or bag-valued attribute or component f of entity type e is given by a pair of integers (low,high). The semantics is given by

($\forall$ v : e) low $\leq$ CNT(f(v)) $\wedge$ CNT(f(v)) $\leq$ high. $\square$

5.9 Example: The example in chapter 2 restricted the number of streets for a district. These kind of constraints cannot be expressed in the "classical" relational calculi. Another such typical example again involves aggregate functions. Domain or tuple calculus cannot express a constraint like "The average age of the ministers of a country should not be greater than 65". If there is a component "ministers : COUNTRY $\rightarrow$ set(PERSON)" and an attribute "age : PERSON $\rightarrow$ int", this constraint reads in our calculus like

$\forall$ c:COUNTRY (AVG -[age(p) | p:PERSON $\wedge$ p IN ministers(c)]- $\leq$ 65).

Acknowledgement

Thanks go to Uwe Hohenstein for our collaboration on the model and the calculus. Special acknowledgement goes to Jugoslavia: For the kindness of the people of "Starigrad Paklenica" and for the uniqueness of the "Plitvicka Jezera" region.

References

[Bi 87] Bird, R.S.: An Introduction to the Theory of Lists. Logic of Programming and Calculi of Discrete Design. Nato ASI Series, Vol. F36, Springer-Verlag Berlin Heidelberg 1987 (M. Broy, ed.) (pp. 5-42)

[Bü 87] Bültzingsloewen, G. v.: Translating and Optimizing SQL Queries Having Aggregates. 13th VLDB, Brighton 1987

[Ce 83] Ceri, S. (ed.): Methodology and Tools for Database Design. North-Holland, Amsterdam 1983

[Ch 76] Chen, P.P.: The Entity-Relationship Model - Towards a Unified View of Data. ACM Transactions on Database Systems, Vol. 1, No. 1, March 1976 (pp. 9-36)

[DMW 82] Dosch, W. / Mascari, G. / Wirsing, M. : On the Algebraic Specification of Databases. Proc. 8th VLDB, 1982

[EDG 86] Ehrich, H.-D. / Drosten, K. / Gogolla, M.: Towards an Algebraic Semantics for Database Specification. Proc. IFIP TC2 Working Conference on Knowledge & Data "DS-2", Albufeira (Portugal), November 1986

[EKW 78] Ehrig, H. / Kreowski, H.-J. / Weber, H. : Algebraic Specification Schemes for Data Base Systems. Proc. 4th VLDB, 1978

[EM 85] Ehrig, H. / Mahr, B.: Fundamentals of Algebraic Specification I. Springer-Verlag, Berlin 1985

[GMS 83] Golshani, F. / Maibaum, T. / Sadler, M. : A Modal System of Algebras for Database Specification and Query/Update Language Support. Proc. 9th VLDB, 1983

[HG 88] Hohenstein, U. / Gogolla, M. : Towards a Semantic View of an Extended Entity- Relationship Model. TU Braunschweig, Informatik-Bericht Nr. 88-02, 1988. Short Version: A Calculus for an Extended Entity-Relationship Model Incorporating Arbitrary Data Operations and Aggregate Functions. Proc. 7th Int. Conf. on Entity- Relationship Approach, North Holland, 1988 (to appear)

[JS 82] Jaeschke, G. / Schek, H.-J.: Remarks on the Algebra of Nonfirst Normal Form Relations. Proc. of the 1st ACM SIGACT-SIGMOD Symp. on Principles of Database Systems. Los Angeles (California), March 1982 (pp. 124-146)

[Kl 82] Klug, A.: Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. Journal of the ACM, Vol. 29, No. 3, July 1982 (pp. 699-717)

[KMS 85] Khosla, S. / Maibaum, T. / Sadler, M. : Database Specification. Proc. IFIP Working Conference on Database Semantics DS-1, 1985

[Ma 83] Maier, D: The Theory of Databases. Computer Sciene Press, Rockville MD 1983

[ÖÖM 87] Özsoyoglu, G. / Özsuyoglu, Z.M. / Matos, V.: Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions. ACM Transactions on Database Systems, Vol. 12, No. 4, December 1987 (pp. 566-592)

[PS 85] Parent, C. / Spaccapietra, S.: An Algebra for a General Entity-Relationship Model. IEEE Transactions on Software Engineering, Vol. 11, No. 7, July 1985 (pp. 634-643)

[Sh 81] Shipman, D.W.: The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems, Vol. 6, No. 1, 1981 (pp. 140-173)

[Su 87] Subieta, K. : Denotational Semantics of Query Languages. Information Systems, Vol. 12, No. 3, 1987 (pp. 69-82)

[SS 77] Smith, J.M. / Smith, D.C.P.: Database Abstractions: Aggregation and Generalization. ACM Transactions on Database Systems, Vol. 2, No. 2, 1977 (pp. 105-133)

[SS 86] Schek, H.-J. / Scholl, M.H.: The Relational Model with Relation-Valued Attributes. Information Systems, Vol. 11, No. 2, 1986 (pp. 137-147)

[SSE 87] Sernadas, A. / Sernadas, C. / Ehrich, H.-D.: Object-Oriented Specification of Databases: An Algebraic Approach. 13th VLDB, Brighton 1987

[TF 82] Teorey, F.J. / Fry, J.P.: Design of Database Structures.Prentice-Hall, Engelwood Cliffs (N.J.), 1982