Towards a Semantic View of an Extended Entity-Relationship Model

MARTIN GOGOLLA
Technische Universität Braunschweig
and
UWE HOHENSTEIN
Siemens AG, Munich

Nearly all query languages discussed recently for the Entity-Relationship (ER) model do not possess a formal semantics. Languages are often defined by means of examples only. The reason for this phenomenon is the essential gap between features of query languages and theoretical foundations like algebras and calculi. Known languages offer arithmetic capabilities and allow for aggregates, but algebras and calculi defined for ER models do not.

This paper introduces an extended ER model concentrating nearly all concepts of known so-called semantic data models in a few syntactical constructs. Moreover, we provide our extended ER model with a formal mathematical semantics. On this basis a well-founded calculus is developed taking into account data operations on arbitrary user-defined data types and aggregate functions. We pay special attention to arithmetic operations, as well as multivalued terms allowing nested queries, in a uniform and consistent manner. We prove our calculus only allows the formulation of safe terms and queries yielding a finite result, and to be (at least) as expressive as the relational calculi.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification techniques*; H.2.1 [**Database Management**]: Logical Design—*Datamodels*, Languages—*Query Languages*

General Terms: Design, Languages, Reliability, Theory

Additional Key Words and Phrases: Abstract data type, aggregate function, calculus, entity-relationship model, formal semantics, relational completeness, safeness, semantic data model

1. INTRODUCTION

It is commonly accepted today [8, 57] that the entire design of a database requires at least four separate but interdependent design steps:

(1) In the *requirements analysis*, the part of the world that is to be modeled

Author's address: M. Gogolla, Technische Universität Braunschweig, Informatik, Abt. Datenbanken, Postfach 33 29, D-3300 Braunschweig, Germany. email:gogolla@infbs.uucp; U. Hohenstein, Siemens AG, Abt. ZFE IS KOM, Postfach 83 09 55, D-8000 München 83, Germany. email:hstein%ztivax@unido.uucp.

ACM Transactions on Database Systems, Vol. 16, No. 3, September 1991, Pages 369–416.

must be thoroughly analyzed to take into account the requests of potential database users.

(2) Using information gained in the previous step, the structure and behavior of the database have to be specified formally in the *conceptual design*.

(3) The resulting system-independent conceptual schema is mapped into a schema of an implemented data model like the relational one. This step, called *logical design*, is necessary to bridge the gap between rich conceptual structures used for conceptual modeling and processable structures for effective system implementation.

(4) Finally, questions of actual database implementation are treated within the *physical design*.

Among these steps, conceptual design plays a central role. To describe the requirements of later database users in a formal and complete manner, so-called semantic data models [31] are needed. But the notion of *semantics* must be regarded with caution in this context, since only few data models possess a proper formal mathematical semantics, such as Chen's Entity-Relationship (ER) model [10], TAXIS [41], IFO [3] or the algebraic approach of Sernadas et al. [55]. The well-tried and widely accepted ER model is often considered as the most appropriate data model, since it captures most of the important phenomena of the real world and expresses them in a natural and easily understandable way. Nevertheless, a number of extensions have been suggested to improve the semantical expressiveness of the model; see, for example, [1, 13, 16, 28, 35]. Based on the ER approach, several high-level query languages are proposed in the recent literature; these languages claim to be more natural than conventional ones, especially relational query languages. The best-known proposals are CABLE [49], CLEAR [40], the natural language-oriented ERROL [38, 40], GORDAS [27, 28], the Executable Language [2], TDL [12], some graphical issues like HIQUEL [60], the QBE-like GQL/ER [62], and a variant of GORDAS [17]. The newer languages are DESPATH [47] incorporating subtype and hierarchy relationships, LAMBDA [61] for the retrieval of structured documents, and the language for geoscientific applications presented by Lipeck and Neumann [35].

In contrast to the increasing number of ER query languages, less interest is paid to semantical foundations, i.e., calculi and algebras. Several ER algebras are proposed [9, 11, 39, 45, 46] in order to define a notion of ER completeness similar to relational completeness (see Maier [37]). All these approaches introduce different notions of completeness, particularly depending on the underlying variant of the ER model. Common to all is the fact that "complete" query languages have to satisfy only minimal requirements. So they do not take into account several features that rich languages should possess, namely data operations (especially arithmetic capabilities) and aggregate functions (like the computation of the minimum or the average). It seems even more strange that almost all ER query languages are not complete with respect to the proposals in the literature.

Another approach to define completeness is presented by Atzeni and Chen [2]. However, they apply a relational calculus. As far as we know Atzeni and

Chen [2] are the first who perceived that there is need for an ER calculus to define precisely the semantics of query languages. Consequently, nearly all ER languages lack a formal semantics. However, the high-level language NETUL [51] is an honorable exception. NETUL is given a denotational semantics [51, 52] based upon formal data structures similar to networks. This approach does not only allow aggregate functions, but also higher constructs like the transitive closure—neither covered by pure relational calculus or algebra.

The objectives of our paper are threefold:

(1) In Section 2 we present an extended ER model (called EER model in the following), providing
   —arbitrary data types for attribute domains,
   —optional and multivalued attributes,
   —a concept for specialization and generalization [53], and
   —complex structured entity types.
(2) Because data models without formal semantics imply query languages, which do not possess precisely defined semantics, we give our EER model a rigorous mathematical semantics in Section 3.
(3) Based on this semantic data model, we define a powerful calculus in Section 4, supporting all concepts of the EER model as well as data operations on the attribute domains and aggregate functions.

In the conclusions we summarize our ideas and outline some further applications of the calculus.

2. THE BASIC IDEA AND A SIMPLE EXAMPLE

Before we present a formal definition of the EER calculus and the underlying EER model, we give a brief informal description of both by means of an example. Therefore, we consider a simplified geoscientific application. We want to store information about different kinds of bodies of water, about countries, towns, and the leaders of these countries and towns. We pay attention to he following facts:

—Every TOWN *lies in* a COUNTRY and may *lie at* one or several RIVERS.
—RIVERS *flow through* COUNTRIES and *flow into* some kind of WATERS.
—SEAS, RIVERS, and LAKES *are* such WATERS.
—A PERSON may be the *mayor of* a TOWN, or the *boss* or a *minister* of a COUNTRY.

We want to consider the usual properties of entities like the *name* of a town, the *population* of a country, or the *list* of *addresses* of a person. Additionally, we associate with every geographic object a geometry in the world's coordinate system:

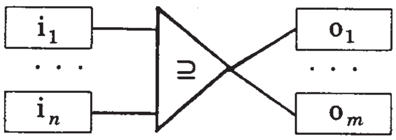
—Countries are represented by a set of closed *polygons* (representing their regions),
—towns and lakes by *circles*,

—seas by closed *polygons*, and
—rivers by connected, nonoverlapping *lines*.

For the conceptual modeling of the part of the world being considered, we use the "semantic" data model proposed by Hohenstein et al. [32]. This approach starts from Chen's (ER) model [10] viewing the universe of discourse as consisting of entities and relationships among them. Both entity and relationship types can have attributes representing their properties. We have extended the original ER model by several concepts, providing a more natural modeling of data abstractions. Therefore, let us consider the EER diagram in the appendix corresponding to the geoscientific part of the world described above.

First of all we identify the basic concepts of the ER model, the entity types PERSON, COUNTRY, TOWN, RIVER, SEA, LAKE, WATERS, the relationship types *is-mayor-of*, *lies-in*, *lies-at*, *flows-through*, *flows-into*, and the attributes *pname*, *addr*, *age* of PERSON, *distance* of *lies-at* (providing the shortest distance between a town and a river), and so on.

—Furthermore, we not only allow standard data types like *int*, *real*, or *string* for attribute domains but also *arbitrary data types*. So we have the geometrical attributes *tgeo*, *rgeo*, and *sgeo* with complex data types *circle*, *lines*, and *polygon*, respectively. The data type *forms* is an enumeration type containing special forms of government (for instance, monarchic, democratic, socialistic, and dictatorial). An *address* consists of a postal number, a city, a street and a street number. Corresponding to the theory of abstract data types, we also consider operations that are allowed on the value domains. Besides arithmetic operations like +, *, for integers, we can define further operations, e.g., to determine the center of a circle or to compute whether two polygons have common points.
—To provide modeling primitives for specialization and generalization known from Smith and Smith [53], we introduce the concept of *type construction*. In our example we have the constructed entity type WATERS, meaning that this entity type is a generalization of SEAS, RIVERS, and LAKES. Consequently, every entity in WATERS is really either a SEA, a RIVER, or a LAKE. The general form of type construction is given by



where $i_1, \ldots, i_n$ are already defined or basic entity types, called *input types*. Starting with these types, the *output types* $o_1, \ldots, o_m$ are constructed. This means that all entities from the input types are put together and are distributed over the output types. Thus, the entities from the output types are not new entities, they already exist (in the input types) but will be now seen in a new context (in one of the output types). We assume every output entity to be in exactly one of the input types. Indicated by the inclusion symbol '$\supseteq$' in the triangle, the inverted direction need not hold, but can be explicitly required by an additional constraint. For example, the generalization of SEA, RIVER, and LAKE to WATERS is constrained in this way, indicated by the equality symbol '='. The concept of type construction is very powerful, since it covers the data abstractions generalization or superclasses ($n > 1$, $m = 1$), specialization or subclasses ($n = 1$, $m = 1$, please beware of the inclusion!), and partition ($n = 1$, $m > 1$), not to mention its general form.

—A quite different extension allows us to describe complex structured entity types (see Lipeck and Neumann [35] and Ramm et al. [48]), whose entities are composed of other entities. For example, each entity of type COUNTRY possesses a *name*, an *area*, a *population*, and a *form of government* consisting of two components, a *boss* and a *list of ministers*, both belonging to the type PERSON. Components can be considered as entity-valued attributes. Of course, the components could belong to different entity types, perhaps again complex structured types.

—As we have just seen for ministers, components and attributes may be *multivalued*. We distinguish between sets, bags (or multisets), and lists. Sets contain an element only once, whereas an element may appear more than once in a bag, i.e., bags retain duplicates of an element. Lists have their elements enumerated so that we can reference them. In our example, the ministers of a country are modeled by a list of people. Thus we can distinguish and access the prime minister (assuming that he or she is the first element in the list). Of course, many other modeling alternatives are possible. Similar to components, attributes can be multivalued; see for example the attribute *addr*. However, we often prefer that the data types (and not the attributes) be multivalued. This enables us to add more data operations, e.g., to determine the *start* point for *lines*.

—To support incomplete information, we introduce *optional* attributes and components, allowing the null value $\perp$ to represent *undefined* or *unknown*. For example, the age of a person may be unknown (indeed, people often "forget" their age). Similarly, it is possible that a country does not possess a boss, i.e., the country has an artificial boss $\perp$. Optional attributes and components are the default. To prohibit optionality, additional constraints can be specified.

For all concepts mentioned above we give a rigorous mathematical semantics, mainly based on the notions of set and function. For reasons of completeness, we mention further concepts that are not considered in this paper because they do not provide any modeling primitives, but rather structural restrictions on the possible database contents. For some motivation, consult Hohenstein et al. [31a, 32].

—Relationships and multivalued attributes or components can be restricted by *cardinalities* as Lipeck and Neumann do [35].

—Relationships can be specified to be *functional*.

—We propose an *extended key concept*, not only allowing attributes as keys (e.g., *tname* of TOWN) but components and functional relationships (key functions presented by Ehrich et al. [14] and Ehrich [15]). The relationship *lies-in* should indeed be functional from TOWN to COUNTRY, since every town lies in exactly one country. This relationship could be used as a further key allowing different towns to have the same name, if they lie in different countries.

—Attributes, components, relationships, and type constructions can be specified as *derived*, meaning that they do not have to be stored explicitly, but can be computed from other stored information. For example, the relationship *lies-in* can be computed from the geometrical attributes of TOWN and COUNTRY.

—The database definition can be completed by explicit *integrity constraints* in order to restrict possible database instances to "consistent" ones.

Taking our EER model as foundation, we now give a first impression of the EER calculus by means of our geoscientific example. This calculus can be considered as a high-level and descriptive query language for the EER model. Our calculus follows the definition of the well-known tuple calculus [37]; however, we prefer the notation of Pirotte [43], for example,

$$\{\!\{ \text{pname}(p) \mid (p : \text{PERSON}) \wedge \exists (c : \text{COUNTRY})\ (\text{cname}(c) = \text{'Italy'} \wedge \exists (t : \text{TOWN})\ (\text{is-mayor-of}(p, t) \wedge \text{lies-in}(c, t)))\}\!\}$$

expresses the query "Give me the names of the mayors of Italian towns." Please note that the result of this query is a bag (or multiset) retaining multiple names if two mayors have the same name. The above terms always have the form

$$\{\!\{ \langle\text{target-terms}\rangle \mid \langle\text{variable-declarations}\rangle \wedge \langle\text{formula}\rangle \}\!\}.$$

Very roughly this can be compared with the SQL *selected-from-where* schema. But, in contrast to SQL, our calculus is completely orthogonal. In particular, we allow these bag-valued terms (in contrast to SQL) to occur in the ⟨target-terms⟩ and ⟨variable-declarations⟩ parts.

Although relational calculus is used to fix the expressiveness of relational query languages defining the notion of "relational completeness," its language features are not sufficient. The relational calculus does not take into account data operations, especially arithmetic operations, or aggregate functions. As far as we know, Klug [34] was the first to propose a calculus incorporating aggregate functions. His calculus was based on the relational model. Because he does not distinguish between sets and bags, his proposal leads to queries that are difficult to understand. This is because the concept of bag is hidden in the one of sets, by adding to the intended information additional data to retain duplicates. A newer proposal by Ozsoyoglu et al. [42] has appeared in the meantime, which extends Klug's approach by using set-valued attributes straightforwardly, without however removing the abovementioned drawback. Thus, we want to design a more readable calculus, paying special attention to the formal and deliberate but comprehensible semantics of data operations and aggregate functions, just as we do to the special concepts of the EER model. To formulate the query "Give me for each stored country its name and the average age of its ministries," we can simply write

$$\{\!\!\{\text{cname(c)}, \text{AVG}\{\!\!\{\text{age(p)} \mid (\text{p}:\text{PERSON}) \wedge \text{p IN ministers(c)}\}\!\!\} \mid (\text{c}:\text{COUNTRY})\}\!\!\}.$$

Please note that it is essential for the computation of the average age that the subquery yield for every country a bag (and not a set!) of integers. Furthermore, the subquery $\{\!\!\{\text{age(p)} \mid \cdots\}\!\!\}$ has a free variable, namely c, and is evaluated for each stored country.

When developing the calculus, we have in mind two properties (apart from a precise semantics analogous to the relational calculus):

(1) The calculus should be *as powerful as the relational calculus* (if every relation is modeled by an entity type).

(2) Every multivalued term, especially every query, must yield a finite result, because properties of infinite sets are not effectively computable. This property is similar to *"safe" calculus expressions* [37] in the relational model.

Both characteristics are proved in Section 4. This makes our calculus usable for several submodels of the ER model, such as the three classical data models (hierachical, network, and relational) and newer proposals like the functional model [50] and the NF^2 model [54].

Even the ER model [10] and its extensions [28] can use the calculus to define the semantics of query languages. What one does is to restrict the EER model to the concepts used in the model at hand. The easiest way for the definition of semantics is to translate the query language into the calculus, as has been done for the relational query language SQL Bültzingsloewen [7]. This way seems more flexible to us than the proposals of Turner and Lowden [58] (for SQL) and Subieta and Missala [51] and Subieta [52] (for other data models, especially the ER model), who map the query language directly into a theory of sets and functions.

Our calculus is particularly suited to formally define the semantics of relational query languages. It can express aggregation features like SUM or AVG (for average) and grouping features like the SQL *group-by* in a natural and understandable way.

3. STRUCTURE OF DATA AND ENTITIES

In this section we present the basic terms relevant for our notion of an entity-relationship schema, where the notion schema refers to the result of one concrete modeling process with our EER model. We put more structure into the schema by strictly distinguishing between a data level and an entity (or object) level. Entities (or objects) are abstract items which are described by attributes. The attribute values are values of data types specified by the schema designer which can be stored in a database. Furthermore, we allow different extensions in addition to the classical ER model, such as means to construct entities from other entities and to structure entities so that they can have other entities as components. For all terms discussed in this section, we give precise mathematical definitions, mainly employing the notions of set and function. Readers unfamiliar with mathematical logic are invited not to go into the details of the definitions on first reading but to take a look at the examples that usually follow the formal definitions. Let us visualize the job of this paper as the job of building a house and let us call it in our case an EER house. The EER model is the foundation, and the EER calculus represents the walls and the roof. The bricks used for the foundation and for the walls and roof are the following "mathematical atoms" such as function, set, bag, and so on.

Axiomatic Conventions

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

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

The syntax of a data signature gives the names of the data sorts (or data types) and the operations and predicates upon them. We assume certain sorts (e.g., int, string) to be predefined. The semantics of a data signature associates a (possibly infinite) set with every sort name, and functions and relations with the operation and predicate names. This interpretation will be fixed once and for all and will not change.

Definition. (Data signature). The *syntax of a data signature DS* is given by

—the sets DATA, OPNS, PRED $\in |\text{FISET}|$,

—a function *source* $\in |\text{FUN}|$ such that *source*: OPNS $\to$ DATA*,

—a function *destination* $\in |\text{FUN}|$ such that *destination*: OPNS $\to$ DATA, and

—a function *arguments* $\in |\text{FUN}|$ such that *arguments*: PRED $\to$ DATA$^+$.

If $\sigma \in$ OPNS, $source(\sigma) = \langle d_1, \ldots, d_n \rangle$, and $destination(\sigma) = d$, this is denoted as $\sigma: d_1 \times \cdots \times d_n \to d$. If $\pi \in$ PRED with $arguments(\pi) = \langle d_1, \ldots, d_n \rangle$, this is denoted as $\pi: d_1 \times \cdots \times d_n$. The *semantics of a data signature DS* is given by

—a function μ[DATA] $\in |\text{FUN}|$ such that μ[DATA]: DATA $\to |\text{SET}|$ and $\perp \in \mu[\text{DATA}](d)$ for every $d \in$ DATA,

—a function μ[OPNS] $\in |\text{FUN}|$ such that μ[OPNS]: OPNS $\to |\text{FUN}|$ and σ:

$d_1 \times \cdots \times d_n \to d$ implies $\mu[\text{OPNS}](\sigma)$: $\mu[\text{DATA}](d_1) \times \cdots \times \mu[\text{DATA}](d_n) \to \mu[\text{DATA}](d)$ for every $\sigma \in \text{OPNS}$, and

—a function $\mu[\text{PRED}] \in |\text{FUN}|$ such that $\mu[\text{PRED}]$: $\text{PRED} \to |\text{REL}|$ and $\pi : d_1 \times \cdots \times d_n$ implies $\mu[\text{PRED}](\pi) \subseteq \mu[\text{DATA}](d_1) \times \cdots \times \mu[\text{DATA}](d_n)$ for every $\pi \in \text{PRED}$.

The set $\mathbf{OPNS^{BIN}}$ denotes all binary operators $\sigma \in \text{OPNS}$ having source(σ) = $\langle d, d \rangle$ and destination(σ) = d for some $d \in \text{DATA}$. Additionally, the corresponding function $\mu[\text{OPNS}](\sigma)$ has to be commutative and associative.

Example. (Predefined data sorts, operators, and predicates). Assumption:

$$
\begin{array}{lll}
\text{DATA} \supseteq \{ \mathit{int}, \mathit{real}, \mathit{string} \} & & \\
\text{OPNS} \supseteq \{ +_i, -_i, *_i: & \mathit{int} \times \mathit{int} & \to \mathit{int}, \\
\quad +_r, -_r, *_r: & \mathit{real} \times \mathit{real} & \to \mathit{real}, \\
\quad /_i: & \mathit{int} \times \mathit{int} & \to \mathit{real}, \\
\quad /_r: & \mathit{real} \times \mathit{real} & \to \mathit{real}, \\
\quad \uparrow_i: & \mathit{int} \times \mathit{int} & \to \mathit{int}, \\
\quad \uparrow_r: & \mathit{real} \times \mathit{int} & \to \mathit{real}, \\
\quad \text{square}_i: & \mathit{int} & \to \mathit{int}, \\
\quad \text{square}_r: & \mathit{real} & \to \mathit{real}, \\
\quad \text{abs}_i: & \mathit{int} & \to \mathit{int}, \\
\quad \text{abs}_r: & \mathit{real} & \to \mathit{real}, \\
\quad \text{trc, rnd}: & \mathit{real} & \to \mathit{int}, \\
\quad \text{cat}: & \mathit{string} \times \mathit{string} & \to \mathit{string}, \\
\quad \text{length}: & \mathit{string} & \to \mathit{int} \} \\
\text{PRED} \supseteq \{ <_i, >_i, \leq_i, \geq_i, \neq_i, =_i: \mathit{int} \times \mathit{int}, & & \\
\quad <_r, >_r, \leq_r, \geq_r, \neq_r, =_r: \mathit{real} \times \mathit{real}, & & \\
\quad <_s, >_s, \leq_s, \geq_s, \neq_s, =_s: \mathit{string} \times \mathit{string} \} & &
\end{array}
$$

These predefined sorts, operations, and predicates have the "usual" semantics (with *real* referring to rational numbers), and this interpretation is fixed once and for all. The semantics can be explicitly specified; for example, algebraically using (conditional) equations (additionally, we then need generators like 0, +1, − 1). For more details concerning algebraic data type specification consult Ehrig and Mahr [18]. One of the nicest features of abstract data type theory is the fact that it provides means for choosing the semantics "canonically."

Throughout the paper, the Greek letter μ stands for "meaning" (and not for "mysterious"). Thus, if we have, for example,

$$\mathit{int} \in \text{DATA},$$

we define the meaning of the symbol *int* by specifying

$$\mu[\text{DATA}](\mathit{int}) := \mathbb{Z} \cup \{ \perp \}.$$

If we additionally have two function symbols

$$\text{abs, square}: \mathit{int} \to \mathit{int} \in \text{OPNS},$$

the meaning of, for instance, square is a function determined by

$$\mu[\text{OPNS}](\text{square})\colon\ \mathbb{Z} \cup \{\perp\} \to \mathbb{Z} \cup \{\perp\}$$
$$i \mapsto \text{if } i \in \mathbb{Z} \text{ then } i * i \text{ else } \perp .$$

Later we will see that this induces syntactically different terms, e.g.,

$$\text{abs}(-4), \text{square}(2) \in \text{TERM}$$

with the same meaning, i.e., with the same evaluation:

$$\mu[\text{TERM}](\text{abs}(-4)) = \mu[\text{TERM}](\text{square}(2)) = 4.$$

Thus μ[DATA], μ[OPNS], etc., are just names for functions giving each appropriate syntactic category a corresponding semantic category. Please note that both the syntax and semantics of a data signature employ the abovementioned "mathematical atoms": DATA $\in$ |FISET|, μ[DATA] $\in$ |FUN|, and so on. These "bricks" are the only things we use in definitions.

Remark. (Disjointness of sets). Throughout the paper, all sets mentioned in definitions have to be disjoint, except when common elements are explicitly allowed. Thus, for example, DATA, OPNS, and PRED are disjoint as well as (the interpretations of) all data sorts with the exception of $\perp$, that is, $\mu[\text{DATA}](d_1) \cap \mu[\text{DATA}](d_2) = \{\perp\}$, if $d_1 \neq d_2$.

The element $\perp$ denotes the undefined value for data sorts. It is a value in every data sort. We have required every data sort d to contain the value $\perp \in \mu[\text{DATA}](d)$, because it is necessary to have an "undefined" value as a result for incorrect applications of operations. Thus, we obtain, for incorrect applications of operations, this special value, for example, $\mu[\text{OPNS}](/_i)(c, 0) = \perp$ for every constant $c \in \mu[\text{DATA}](d)$.

In most cases, it is reasonable to define the propagation of $\perp$ in the following way: for an operation $\sigma \in \text{OPNS}$ with $\sigma\colon d_1 \times \cdots \times d_n \to d$, $\mu[\text{OPNS}](\sigma)(c_1, \ldots, c_n)$ evaluates to $\perp$ if there is a c_i with $c_i = \perp$. For predicates, $\pi \in \text{PRED}$ with $\pi\colon d_1 \times \cdots \times d_n (c_1, \ldots, c_n) \in \mu[\text{PRED}](\pi)$ does not hold if there is a c_i with $c_i = \perp$.

The operators $/_i\colon$ *int* $\times$ *int* $\to$ *real* and $/_r\colon$ *real* $\times$ *real* $\to$ *real* are similar but different operators. We omit the indices i and r if they are clear from the context. We do the same for other operators and predicates.

The operations $\sigma\colon d_1 \times \cdots \times d_n \to d$ are sometimes used in infix position instead of $\sigma(c_1, \ldots, c_n)$; for example, $c_1 + c_2$ instead of $+(c_1, c_2)$.

In the context of data types (and later on even in the context of entity types), certain standard constructions appear quite often. It makes sense therefore to define these parameterized data types once and use them for several cases. Throughout the paper, we need set-, list-, and bag-valued terms and operations. For this reason, the syntax of the sort expressions allows us to build arbitrary new sorts (e.g., list(set(*int*))), starting with data sorts or other given sorts. The semantics gives a corresponding set for every sort expression.

Definition. (Sort expressions). Let a data signature DS and a set S $\in$ |SET| with DATA $\subseteq$ S together with a (semantic) function $\mu[\text{S}]\colon \text{S} \to |\text{SET}|$ (such that $\mu[\text{S}](\text{d}) = \mu[\text{DATA}](\text{d})$ for $\text{d} \in \text{DATA}$ and $\perp \in \mu[\text{S}](\text{s})$ for every $\text{s} \in \text{S}$) be given. The *syntax of the sort expressions over S* is given by the set EXPR(S) determined by the following rules.

(i) $s \in S$, then $\mathbf{s} \in$ EXPR(S).
(ii) If $s \in$ EXPR(S), then **set(s)** $\in$ EXPR(S).
(iii) If $s \in$ EXPR(S), then **list(s)** $\in$ EXPR(S).
(iv) If $s \in$ EXPR(S), then **bag(s)** $\in$ EXPR(S).
(v) If $s_1, \ldots, s_n \in$ EXPR(S) ($n \geq 1$), then $\mathbf{prod(s_1, \ldots, s_n)} \in$ EXPR(S).

The *semantics of the sort expressions* is a function $\mu[\text{EXPR(S)}]$: EXPR(S) $\to$ |SET| determined by the following rules.

(i) $\mu[\text{EXPR(S)}](s) := \mu[S](s)$.
(ii) $\mu[\text{EXPR(S)}](\text{set}(s)) := F(\mu[\text{EXPR(S)}](s)) \cup \{\perp\}$.
(iii) $\mu[\text{EXPR(S)}](\text{list}(s)) := (\mu[\text{EXPR(S)}](s))^* \cup \{\perp\}$.
(iv) $\mu[\text{EXPR(S)}](\text{bag}(s)) := B(\mu[\text{EXPR(S)}](s)) \cup \{\perp\}$.
(v) $\mu[\text{EXPR(S)}](\text{prod}(s_1, \ldots, s_n)) := (\mu[\text{EXPR(S)}](s_1) \times \cdots \times \mu[\text{EXPR(S)}](s_n)) \cup \{\perp\}$.

$\hat{\mu}[\text{EXPR(S)}]$ is an abbreviation for $\bigcup_{s \in \text{EXPR(S)}} \mu[\text{EXPR(S)}](s)$, i.e., the set of all instances belonging to the sort expressions over the set S.

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. (From the view-point of algebraic specification this is clumsy, but it does the job for this paper.)

$\text{EMPTY}_{\text{set(s)}}: \to \text{set(s)}$
$\text{ADD}_{\text{set(s)}}: \text{set(s)} \times s \to \text{set(s)}$
$\text{EMPTY}_{\text{bag(s)}}: \to \text{bag(s)}$
$\text{ADD}_{\text{bag(s)}}: \text{bag(s)} \times s \to \text{bag(s)}$
$\text{EMPTY}_{\text{list(s)}}: \to \text{list(s)}$
$\text{ADD}_{\text{list(s)}}: \text{list(s)} \times s \to \text{list(s)}$
$\text{MAKE}_{\text{prod}(s_1, \ldots, s_n)}: s_1 \times \cdots \times s_n \to \text{prod}(s_1, \ldots, s_n)$

Additionally, error constants like $\text{ERROR}_{\text{list(s)}}: \to \text{list(s)}$ 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).

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

Because the second set equation is not valid for bags, bags and sets satisfy the following rules (with respect to the $\{\{\cdots\}\}$- and $\{\cdots\}$-notation):

$$\{\{c_1\}\} \cup \{\{c_2\}\} = \{\{c_2\}\} \cup \{\{c_1\}\} = \{\{c_1, c_2\}\} = \{\{c_2, c_1\}\} \quad \text{if } c_1 \neq c_2$$
$$\{\{c\}\} \cup \{\{c\}\} = \{\{c, c\}\}$$
$$\{c_1\} \cup \{c_2\} = \{c_2\} \cup \{c_1\} = \{c_1, c_2\} = \{c_2, c_1\} \quad \text{if } c_1 \neq c_2$$
$$\{c\} \cup \{c\} = \{c\}$$

Example. (Specification of data types). The following lines can (roughly speaking) be understood as a definition of the nonstandard data types used in our example.

```
sorts        point, circle, polygon, region, forms, address, lines, regions
definitions  point   = prod (real,real)
             circle  = prod (point,real)
             address = prod (int,string,string,int)
             polygon = list (point)
             lines   = list (point)
             region  = prod (string,polygon)
             regions = set (region)
operations   x: point → real                  /* x and y coordinate
             y: point → real                  /* of a point
             pdist: point × point → real      /* distance of two points
             start: lines → point             /* start and end point
             end: lines → point               /* of (connected) lines
             center: circle → point           /* center and radius
             radius: circle → real            /* of a circle
             name: region → string            /* name and geometry
             geo: region → polygon            /* of a region
             monarchic, democratic, socialistic, dictatorial: → forms
                                              /* forms of government
             pno: address → int               /*
             city: address → string           /* components of
             street: address → string         /* an address
             no: address → int                /*
             ...
predicates   ppcut: polygon × polygon         /* is satisfied iff two polygons,
             lpcut: lines × polygon           /* lines and a polygon, or a circle and
             cpcut: circle × polygon          /* a polygon have common points
             ...
...
```

The defined sorts can be viewed as new sorts with generators according to the corresponding sort expressions. For instance, if a sort is introduced via

definitions newsort = $\text{oldsort}_1 \times \cdots \times \text{oldsort}_n$,

this corresponds to a generator of the form

$\text{MAKE}_{\text{newsort}}: \text{oldsort}_1 \times \cdots \times \text{oldsort}_n \rightarrow \text{newsort}.$

On the other hand, if a sort is defined by

definitions newsort = [list | set | bag] (oldsort),

this corresponds to generators of the form

$\text{EMPTY}_{\text{newsort}}: \rightarrow \text{newsort}$
$\text{ADD}_{\text{newsort}}: \text{newsort} \times \text{oldsort} \rightarrow \text{newsort}.$

Additionally, equations have to be added for sets and bags. We can understand the type *point* as a structured record type consisting of an x- and a y-coordinate. The function *pdist* gives the distance between two points. Similar to the record structure of a point, a *circle* has a center and a radius; a *region* has a *name* and a *geometry*; and an *address* is structured into *pno* (postal number), *city*, *street*, and *no*. The type *forms* is not constructed, but an enumeration type containing (here) the values *monarchic*, *democratic*, *socialistic*, and *dictatorial*. The types *lines*, *polygon*, and *regions* are constructed using set and list expressions. *lines* and *polygon* are both constructed by list(*point*). We assume that *lines* represent connected lines, and thus have a *start* and an *end* point, both different from each other. On the other hand, *polygon* stands for closed lines without *start* and *end* points.

In general, it is not sufficient to have new sorts, but we want to deal with properties of these bags, lists, and so on. Therefore, we introduce standard aggregate operations and predicates. For example, we provide a way to count the number of elements in a set, to apply an operation (similar to Bird [4]) to a set of values (e.g., the application of + to a set of integers corresponds to building the sum), to convert a list to a bag, and so forth, please remember that the set $\mathrm{OPNS}^{\mathrm{BIN}}$ typically includes operators like + (for the addition of numbers), but not operators like − or /, because their corresponding interpretation is not commutative. $\mathrm{OPNS}^{\mathrm{BIN}}$ is used to formally define aggregations like SUM or AVG.

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. ($s, s_1, \ldots, s_n$ refer to arbitrary sort expressions and σ to an element of $\mathrm{OPNS}^{\mathrm{BIN}}$ with $\sigma\colon d \times d \to d$).

OPNS(S)/PRED(S)	Source/arguments	Destination	Informal description
$\mathrm{CNT}_{\mathrm{set(s)}}$	: set(s)	$\to$ *int*	Counts the elements
$\mathrm{IND}_{\mathrm{set(s)}}$	: set(s)	$\to$ set(*int*)	Set of indices of a set
$\mathrm{APL}_{\sigma,\mathrm{set(d)}}$	: set(d)	$\to$ d	Applies a binary operation to a set
$\mathrm{IN}_{\mathrm{set(s)}}$	: set(s) $\times$ s		Element-of relation
$\mathrm{CNT}_{\mathrm{list(s)}}$	: list(s)	$\to$ *int*	Counts the elements
$\mathrm{IND}_{\mathrm{list(s)}}$	: list(s)	$\to$ set(*int*)	Set of indices of a list
$\mathrm{LTB}_{\mathrm{list(s)}}$	: list(s)	$\to$ bag(s)	Converts a list to a bag
$\mathrm{SEL}_{\mathrm{list(s)}}$	: list(s) $\times$ int	$\to$ s	Selects the *i*th element
$\mathrm{POS}_{\mathrm{list(s)}}$	: list(s) $\times$ s	$\to$ set(*int*)	Set of indices of an element
$\mathrm{APL}_{\sigma,\mathrm{list(d)}}$	: list(d)	$\to$ d	Applies a binary operation to a list
$\mathrm{IN}_{\mathrm{list(s)}}$	: list(s) $\times$ s		Element-of relation
$\mathrm{CNT}_{\mathrm{bag(s)}}$	: bag(s)	$\to$ *int*	Counts the elements
$\mathrm{IND}_{\mathrm{bag(s)}}$	: bag(s)	$\to$ set(*int*)	Set of indices of a bag
$\mathrm{BTS}_{\mathrm{bag(s)}}$	: bag(s)	$\to$ set(s)	Converts a bag to a set
$\mathrm{OCC}_{\mathrm{bag(s)}}$	: bag(s) $\times$ s	$\to$ *int*	Counts the occurrences
$\mathrm{APL}_{\sigma,\mathrm{bag(d)}}$	: bag(d)	$\to$ d	Applies a binary operation to a bag
$\mathrm{IN}_{\mathrm{bag(s)}}$	: bag(s) $\times$ s		Element-of relation
$\mathrm{PRJ}_{\mathrm{prod}(s_1,\ldots,s_n),\,i}$	: prod$(s_1, \ldots, s_n)$	$\to s_i$ $(i \in 1..n)$	Projection

The *semantics of the operations induced by the sort expressions* is a function $\mu[\mathrm{OPNS(S)}]\colon \mathrm{OPNS(S)} \to |\mathrm{FUN}|$ and a function $\mu[\mathrm{PRED(S)}]\colon \mathrm{PRED(S)} \to |\mathrm{REL}|$ 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 $\mu[\text{OPNS(S)}](\sigma)$: $\mu[\text{EXPR(S)}](s_1) \rightarrow \mu[\text{EXPR(S)}](s_2)$, and if $\pi : s$, then $\mu[\text{PRED(S)}](\pi) \subseteq \mu[\text{EXPR(S)}](s)$, where s refers to a sort expression (analogously for functions and predicates with two arguments). Furthermore, we abbreviate $\mu[\text{OPNS(S)}](\sigma)$ by $\mu(\sigma)$ and $\mu[\text{PRED(S)}](\pi)$ by $\mu(\pi)$. All these functions preserve the undefined value $\perp$ ($\mu(\sigma)(\perp) = \perp$ and $\mu(\sigma)(c_1, c_2) = \perp$, if $c_1 = \perp$); $c, c_1, \ldots, c_n$ are always elements of $\mu[\text{EXPR(SORT)}](s)$ resp., $\mu[\text{EXPR(SORT)}](d)$.

$$\mu(\text{CNT}_{\text{set(s)}}): \{c_1, \ldots, c_n\} \mapsto n$$

$$\mu(\text{IND}_{\text{set(s)}}): \{c_1, \ldots, c_n\} \mapsto \{1, \ldots, n\}$$

$$\mu(\text{APL}_{\sigma, \text{set(d)}}):$$

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

$$\mu(\text{IN}_{\text{set(s)}}) := \{(\{c_1, \ldots, c_n\}, c) \mid c \in \{c_1, \ldots, c_n\}\}$$

$$\mu(\text{CNT}_{\text{list(s)}}): \langle c_1, \ldots, c_n \rangle \mapsto n$$

$$\mu(\text{IND}_{\text{list(s)}}): \langle c_1, \ldots, c_n \rangle \mapsto \{1, \ldots, n\}$$

$$\mu(\text{LTB}_{\text{list(s)}}): \langle c_1, \ldots, c_n \rangle \mapsto \{\{c_1, \ldots, c_n\}\}$$

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

$$\mu(\text{POS}_{\text{list(s)}}):$$

$$(\langle c_1, \ldots, c_n \rangle, c) \mapsto \begin{cases} \varnothing & \text{if } n = 0 \\ \mu(\text{POS}_{\text{list(s)}})(\langle c_1, \ldots, c_{n-1} \rangle, c) \cup \{n\} & \text{if } n \geq 1 \text{ and } c_n = c \\ \mu(\text{POS}_{\text{list(s)}})(\langle c_1, \ldots, c_{n-1} \rangle, c) & \text{if } n \geq 1 \text{ and } c_n \neq c \end{cases}$$

$$\mu(\text{APL}_{\sigma, \text{list(d)}}):$$

$$\langle c_1, \ldots, c_n \rangle \mapsto \begin{cases} \perp & \text{if } n = 0 \\ c_1 & \text{if } n = 1 \\ \mu(\sigma)(c_1, \mu(\text{APL}_{\sigma, \text{list(d)}})(\langle c_2, \ldots, c_n \rangle)) & \text{if } n \geq 2 \end{cases}$$

$$\mu(\mathrm{IN}_{\mathrm{list(s)}}) := \{(\langle c_1, \ldots, c_n \rangle, c) \mid c \in \{c_1, \ldots, c_n\}\}$$

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

$$\mu(\mathrm{BTS}_{\mathrm{bag(s)}}):$$

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

$$\mu(\mathrm{OCC}_{\mathrm{bag(s)}}):$$

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

$$\mu(\mathrm{APL}_{\sigma, \mathrm{bag(d)}}):$$

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

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

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

In the above definitions, $\{\ldots\} \cup \{\ldots\}$ refers to the union of sets (this guarantees the elimination of duplicates, e.g., $\{c\} \cup \{c\} = \{c\}$), and $\{\{\ldots\}\} \cup \{\{\ldots\}\}$ refers to the union of bags respecting duplicates, e.g., $\{\{c\}\} \cup \{\{c\}\} = \{\{c, c\}\}$). Equational algebraic specifications can be given for all presented functions.

Notation. (Conversions and aggregation functions). We use the following conventions and abbreviations for frequently used operations. If no ambiguities occur, we drop the list-, set-, bag-, prod-indices of the operation symbols and leave out parentheses in accordance with the usual rules. **LTS(x)** is equivalent to BTS(LTB(x)) and converts a list to a set. **SUM** refers to APL_{+}, depending on the context. **MAX** means $\mathrm{APL}_{\max}$, depending on the context, with $\max(x, y) = \mathit{if}\ x > y\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{fi}$. **MIN** is defined analogously. $\mathbf{AVG}_{\mathrm{set(d)}}(x)$ refers to $\mathrm{APL}_{+, \mathrm{set(d)}}(x) / \mathrm{CNT}_{\mathrm{set(d)}}(x)$, and analogously for lists and bags. With this convention we have, of course, AVG(∅) = ⊥ . Instead of PRJ(x) or $PRJ_i((x_1, \ldots, x_n))$, we also use the more suggestive **x.i** or $(x_1, \ldots, x_n)$.i, where i is a constant between 1 and n. Apart from standard aggregates like AVG or SUM, one can also define aggregates concerning user-defined data types, e.g., the center of gravity, COG, for a set of points.

$$\text{COG: set(point)} \mapsto \text{point}$$
$$\{p_1, \ldots, p_n\} \mapsto (\text{AVG}\{x(p_1), \ldots, x(p_n)\}, \text{AVG}\{y(p_1), \ldots, y(p_n)\})$$

Remark. With this convention, our aggregate functions SUM, MAX, and MIN do not come out of the blue, but are formally defined.

Example. (Aggregate functions). To become a bit more familiar with the (perhaps unconventional) notion *bag*, we give a few simple examples involving aggregate functions.

$$\text{SUM}\,\{\{1,2,1,3,2\}\} = \text{SUM}\,\{\{3,2,2,1,1\}\} = 9$$
$$\text{SUM}\,(\text{BTS}\,\{\{1,2,1,3,2\}\}) = \text{SUM}\,(\text{BTS}\,\{\{1,1,2,2,3\}\})$$
$$= \text{SUM}\,\{1,2,3\} = 6$$
$$\text{AVG}\,\{\{1,2,2,3,1\}\} = \text{SUM}\,\{\{1,2,2,3,1\}\}/\text{CNT}\,\{\{1,2,2,3,1\}\} = 1.8$$
$$\text{AVG}\,\{1,2,3\} = \text{AVG}\,(\text{BTS}\,\{\{1,2,2,3,1\}\}) = 2$$
$$\text{AVG}\,\langle 1,2,3,2,1\rangle = \text{AVG}\,(\text{LTB}\langle 1,2,3,2,1\rangle) = 1.8$$

We now come to the central notion of this section: the extended entity-relationship schema (the EER schema). The syntax of such a schema introduces names for the entity (or object) types, for the relationship types between the entity types, and for the attributes (or properties) of entities. Such a schema also offers the possibility that entities can be components of other entities and that entities can be constructed from other entities. The semantics of an EER schema associates sets and functions with the things mentioned above. For instance, for every entity type there is a (finite) set of current entities, and for every attribute name there is a function giving for every entity the current value of the attribute. These sets and functions constitute a (database) state, which can be altered while time changes (in contrast to the interpretation of the data types, which is fixed). The connection to the data signature is established via the attributes that are data-valued functions.

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 ENTTYPE, RELTYPE, ATTRIBUTE, COMPONENT, CONSTRUCTION ∈ |FISET| and

—the functions *participants*, *asource*, *adestination*, *csource*, *cdestination*, *input*, *output* ∈ |FUN| such that

participants: $\text{RELTYPE} \to \text{ENTTYPE}^+$,

asource:	ATTRIBUTE $\rightarrow$ ENTTYPE $\cup$ RELTYPE
adestination:	ATTRIBUTE $\rightarrow$ {d, set(d), list(d), bag(d) $\mid$ d $\in$ DATA},
csource:	COMPONENT $\rightarrow$ ENTTYPE,
cdestination:	COMPONENT $\rightarrow$ {e, set(e), list (e), bag(e) $\mid$ e $\in$ ENTTYPE},
input:	CONSTRUCTION $\rightarrow$ $\mathcal{F}$(ENTTYPE)-$\emptyset$, and
output:	CONSTRUCTION $\rightarrow$ $\mathcal{F}$(ENTTYPE)-$\emptyset$.

If r $\in$ RELTYPE with *participants*(r) = $\langle e_1, \ldots, e_n \rangle$, this is denoted as $r(e_1, \ldots, e_n)$. If a $\in$ ATTRIBUTE with *asource*(a) = e or *asource*(a) = r and *adestination*(a) = d, this is denoted as a: e $\rightarrow$ d or a: r $\rightarrow$ d, respectively. If c $\in$ COMPONENT with *csource*(c) = e and *cdestination*(c) = e′, this is denoted as c: e $\rightarrow$ e′. If c $\in$ CONSTRUCTION with *input*(c) = $\{i_1, \ldots, i_n\}$ and *output*(c) = $\{o_1, \ldots, o_m\}$, this is denoted as $c(i_1, \ldots, i_n; o_1, \ldots, o_m)$.

For two distinct constructions c_1, c_2 $\in$ CONSTRUCTION, the following conditions must hold:

(i) *output*(c_1) $\cap$ *output*(c_2) = $\emptyset$

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

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

—a function μ[ENTTYPE] $\in$ |FUN| such that μ[ENTTYPE]: ENTTYPE $\rightarrow$ |FISET| and $\perp \in \mu$[ENTTYPE](e) for every e $\in$ ENTTYPE;

—a function μ[RELTYPE] $\in$ |FUN| such that μ[RELTYPE]: RELTYPE $\rightarrow$ |FISET|, and $r(e_1, \ldots, e_n)$ implies μ[RELTYPE](r) $\subseteq$ (μ[ENTTYPE](e_1)-$\{\perp\} \times \cdots \times \mu$[ENTTYPE]($e_n$)-$\{\perp\}) \cup \{\perp\}$, and $\perp \in \mu$[RELTYPE](r) for every r $\in$ RELTYPE;

—a function μ[ATTRIBUTE] $\in$ |FUN| such that μ[ATTRIBUTE]: ATTRIBUTE $\rightarrow$ |FUN| and a: e $\rightarrow$ d implies μ[ATTRIBUTE](a): μ[ENTTYPE](e) $\rightarrow$ μ[EXPR(DATA)](d), and a: r $\rightarrow$ d implies μ[ATTRIBUTE](a): μ[RELTYPE](r) $\rightarrow$ μ[EXPR(DATA)](d);

—a function μ[COMPONENT] $\in$ |FUN| such that μ[COMPONENT]: COMPONENT $\rightarrow$ |FUN| and c: e $\rightarrow$ e′ implies μ[COMPONENT](c): μ[ENTTYPE](e) $\rightarrow$ μ[EXPR(ENTTYPE)](e′);

—a function μ[CONSTRUCTION] $\in$ |FUN| such that μ[CONSTRUCTION]: CONSTRUCTION $\rightarrow$ |FUN| and $c(i_1, \ldots, i_n; o_1, \ldots, o_m)$ implies μ[CONSTRUCTION](c): $\bigcup_{j=1}^{m} \mu$[ENTTYPE](o_j) $\rightarrow \bigcup_{k=1}^{n} \mu$[ENTTYPE]($i_k$), where 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.

Example. (EER schema). Let us now consider the simple geographic database application already mentioned, where countries, towns, and special kinds of bodies of water are described. The following schema definition (denoted in a linear way) belongs to our geoscientific application.

```
schema COUNTRY-TOWN-WATERS
enttypes      COUNTRY        attributes     cname         : string,
                                            area          : regions,
                                            cpopulation   : int,
                                            government    : forms;
                             components     boss          : PERSON,
                                            ministers     : list(PERSON);
              TOWN           attributes     tname         : string,
                                            tgeo          : circle,
                                            tpopulation   : int;
              RIVER          attributes     rname         : string,
                                            rgeo          : lines;
              LAKE           attributes     lname         : string,
                                            lgeo          : circle;
              SEA            attributes     sname         : string,
                                            sgeo          : polygon;
              WATERS         attributes     wname         : string;
              PERSON         attributes     pname         : string,
                                            addr          : list(address),
                                            age           : int;
reltypes      lies-in                  participants  COUNTRY, TOWN;
              lies-at                  participants  TOWN, RIVER;
                                       attributes    distance : real;
              flows-through            participants  COUNTRY, RIVER;
                                       attributes    length : real;
              flows-into               participants  RIVER, WATERS;
              is-mayor-of              participants  PERSON, TOWN;
constructions are input   SEA, LAKE, RIVER;
                  output  WATERS;
```

The elements of ENTTYPE are called *entity* (or object) *types*, the elements of RELTYPE are the *relationship types*, and the elements of ATTRIBUTE are the *attributes names*, known from the ER model defined by Chen [10]. μ[ENTTYPE](e) is the set of entities (or instances) belonging to the entity type e, μ[RELTYPE](r) defines which entities are related by r, and μ[ATTRIBUTE](a) gives the attributes of entities or relationships (related entities). Therefore, we have the following situation in our running example: ENTTYPE = {COUNTRY, TOWN, RIVER, ...}, RELTYPE = {lies-in, lies-at, flows-through, ...}, ATTRIBUTE = {pname, addr, age, cname, ...} and so on. The semantics looks like this: μ[ENTTYPE](COUNTRY) = $\{c_1, \ldots, c_k\} \cup \{\perp\}$, μ[RELTYPE](lies-in) $\subseteq (\{t_1, \ldots, t_l\} \times \{c_1, \ldots, c_k\}) \cup \{\perp\}$, μ[ATTRIBUTE](cname): $\{c_1, \ldots, c_k\} \rightarrow A^* \cup \{\perp\}$ (= μ[DATA](*string*)), μ[ATTRIBUTE](cname): $c_1 \mapsto$ 'France', and so on.

Additionally, we have a set COMPONENT of *component names* to model complex entity types as do Lipeck and Neumann [35] and Ramm et al. [48]. 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. Thus, in our running example we have COMPONENT = {boss, ministers} and the semantics μ[COMPONENT](ministers): $\{c_1, \ldots, c_k\} \rightarrow \{p_1, \ldots, p_m\}^*$.

One can also further restrict the possible interpretations of COMPONENT functions in the following way. Let e_1', e_2' be two entities, i.e., $e_1', e_2' \in \bigcup_{e \in \mathrm{ENTTYPE}} \mu[\mathrm{ENTTYPE}](e)$. We say entity e_1' is a direct component of entity e_2', i.e., direct-component(e_1', e_2') holds iff there is a $c \in$ COMPONENT such that $e_1' = \mu[\mathrm{COMPONENT}](c)(e_2')$ or $e_1' \in \mu[\mathrm{COMPONENT}](c)(e_2')$ if c is multivalued. Then it is not allowed that there is an entity e' such that direct-component$^+(e', e')$ holds, where direct-component$^+$ is the transitive closure of direct-component. In other words, it is forbidden for an entity to be a component of itself. Relations between entity types that require these kinds of reflexive properties have to be modeled with relationships and not with components.

To provide modeling primitives for specialization and generalization [53], 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 nonconstructed or already defined entity types in *input*(c), the new entity types in *output*(c) are constructed. The concept of type construction is very powerful, since it covers data abstraction generalizations or superclasses (n > 1, m = 1), specialization or subclasses (n = 1, m = 1, please beware of the inclusion!), and partition (n = 1, m > 1), not to mention its general form.

In our example (see Section 2 and the Appendix) we have CONSTRUCTION = {are}, input(are) = {SEA, LAKE, RIVER} and output(are) = {WATERS}. This construction "are" is a generalization. Other construction types are possible, e.g., CONSTRUCTION = {are, partition-person, specialize-to-port} with input(partition-person) = {PERSON}, output(partition-person) = {MALE, FEMALE}, input(specialize-to-port) = {TOWN}, and output(specialize-to-port) = {PORT-TOWN}. Thus "are" is a generalization, "partition-person" represents a partitioning of the input type "PERSON" and "specialize-to-port" corresponds to a specialization. Although all introduced entity types are disjoint 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.

$$\mu[\mathrm{CONSTRUCTION}](c)\colon \bigcup_{j=1}^{m} \mu[\mathrm{ENTTYPE}](o_j) \to \bigcup_{k=1}^{n} \mu[\mathrm{ENTTYPE}](i_k),$$

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. The injective function $\mu[\mathrm{CONSTRUCTION}](c)$ is a technical requirement, because we want all entity types to be disjoint (with the exception of $\perp$). This semantics of CONSTRUCTION is equivalent to demanding that every output entity "is" an input entity:

$$\bigcup_{j=1}^{m} \mu[\mathrm{ENTTYPE}](o_j) \subseteq \bigcup_{k=1}^{n} \mu[\mathrm{ENTTYPE}](i_k).$$

Furthermore, we do not have the optionality property for CONSTRUCTION, i.e., every output entity (except $\perp$) must be constructed from an input entity different from $\perp$, because each μ[CONSTRUCTION](c) has to be injective. To preserve the intuitive semantics of Hohenstein et al [32], we have restricted the syntactical form of type constructions in the definition in the following way:

(i) Every constructed entity type has to be the result of exactly one type construction.
(ii) Every constructed type must not, directly or indirectly, be an input type of its own type construction, i.e., the directed graph consisting of all entity types as nodes and type constructions as edges must be acyclic.

Since the functions *input* and *output* yield sets of entity types, each two input and each two output types must be different.

These conditions guarantee that we have for a constructed entity e uniquely determined constructions $c_1, \ldots, c_n$, other constructed entities $e_1, \ldots, e_{n-1}$ and a nonconstructed (basic) entity e_n, such that

$$e \xrightarrow{\mu(c_1)} e_1 \xrightarrow{\mu(c_2)} e_2 \xrightarrow{\mu(c_3)} e_3 \cdots e_{n-1} \xrightarrow{\mu(c_n)} e_n.$$

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

Example. As an example of a type construction (in this case a specialization) with more than one output type, consider vehicles, especially cars or ships.

constructions special-vehicles **input** VEHICLE
output CAR, SHIP

Please note that this situation does not in general exclude (for example) planes from being vehicles, because we will only have an injective (and not surjective) function

μ[CONSTRUCTION](special-vehicles):
μ[ENTTYPE](CAR) $\cup$ μ[ENTTYPE](SHIP) $\rightarrow$ μ[ENTTYPE](VEHICLE).

It is also possible that one entity type participates in more than one type construction as an input type. For instance, one can have CONSTRUCTION = {specialize-to-port, specialize-to-capital} with input(specialize-to-port) = input(specialize-to-capital) = {TOWN} and output(specialize-to-port) = {PORT-TOWN} and output(specialize-to-capital) = {CAPITAL}. Then it is not forbidden to have towns that have ports and are capitals.

As for DATA, we assume the element $\perp$ to be an element of every entity type in ENTTYPE. Thus, every attribute and component is optional by default. Indeed, this can be excluded by additional integrity constraints. Due to orthogonality, we include the undefined value in every relationship type, too.

Note, as before, that the syntax and semantics of an EER schema are expressed in terms of the previously mentioned "mathematical atoms" (or "bricks" for our EER house): ENTTYPE $\in$ |FISET|, μ[ENTTYPE] $\in$ |FUN|, etc. The semantics of an EER schema is often called a *state* belonging to the schema. Therefore STATE refers to (ENTTYPE, RELTYPE, ATTRIBUTE, COMPONENT, CONSTRUCTION) and μ[STATE] to (μ[ENTTYPE], μ[RELTYPE], μ[ATTRIBUTE], μ[COMPONENT], μ[CONSTRUCTION]). We could also have defined a class π[STATE] of all *possible states*.

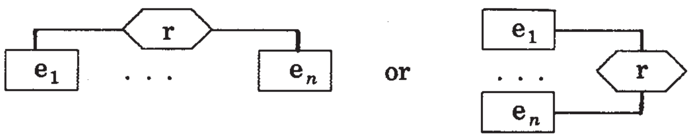
It may be the case that an entity type participates more than once in a relationship. To distinguish the different roles these types play in the relationship, each occurrence of such a participating type can be identified by a rolename.

Definition. (Rolenames). Let an EER schema EER(DS) over DS and a set NAMES of names be given. A *role* is a function *roles* $\in$ |FUN| such that *roles*: RELTYPE $\rightarrow$ (NAMES $\cup$ {*unspecified*})$^+$ and $r(e_1, \ldots, e_m) \in$ RELTYPE implies *roles*(r) = $\langle n_1, \ldots, n_m \rangle$, where $n_i \neq n_j$ for $i \neq j$ (except *unspecified*). If the rolename n_i ($i = 1..m$) for $r(e_1, \ldots, e_m) \in$ RELTYPE is unspecified and if the entity sort e_i appears only once in $\langle e_1, \ldots, e_m \rangle$, we assume e_i is used as the rolename. Rolenames have to be specified if one entity sort participates more than once in a relationship.

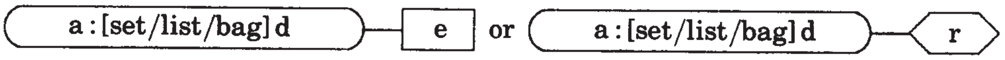
For an EER schema, there exists a graphical representation called its entity-relationship *diagram*. For every syntactical item of an EER schema, a special symbol is introduced. The equivalence is as follows.

—$e \in$ ENTTYPE is represented by the rectangle $\boxed{e}$.

—$r(e_1, \ldots, e_n) \in$ RELTYPE is represented by a diamond connecting the participating entity types. More precisely, the graphical representation has to reflect the order of the participating entity types. We assume the order from left to right and top to bottom.

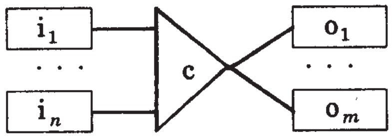


—a: $e \rightarrow d$, a: $r \rightarrow d \in$ ATTRIBUTE are represented by an oval (with the attribute name and the corresponding data type) connected to the entity rectangle or the relationship diamond.



—c: $e \rightarrow e'$/set/list/bag(e') $\in$ COMPONENT are represented by an oval similar to attributes; but instead of the data type we write a square with an arrow, pointing to the corresponding entity type rectangle into the oval.

—$c(i_1, \ldots, i_n; o_1, \ldots, o_m) \in$ CONSTRUCTION is represented by a triangle connecting the input types at the base line and the output types at the opposite point.



A graphical representation of our example is given in the Appendix.

We want to conclude the discussion of our EER model by comparing it to other well-known semantic data models, especially SDM [31], SHM$^+$ [6], and IFO [3]. However, we refrain from an in-depth comparison and an extensive discussion of semantic data models. For such a discussion, consult one of the survey articles in the literature, for example [30, 56]. Our aim is to show how the main concepts, common to these data models, are integrated in the EER model.

When considering the community of semantic data models, some essential concepts appear quite often [56], and should be given attention—namely, aggregation, association, and specialization. Aside from data models, these modeling primitives play an important role in related fields such as knowledge representation [5, 29]. In the following, we want to concentrate on aspects of data models only. Let us therefore attend to the particular concepts.

The first concept mentioned above is aggregation. Generally, aggregation is used for formulating part-of or property-of relationships. Some given (data or entity) types are aggregated to a new type, which possesses the former types as parts. We recognize two forms of aggregation in the "classical" ER model:

—Entity types are aggregations of data types given by attributes, i.e., attributes reflect the property-of relationship.
—Relationship types aggregations of the participating entity types, thus these types are the usual form to model part-of relationships.

The ER model as well as several extensions [28, 59] take on these forms almost without change. But we offer a more general form of aggregation by using (possibly multivalued) components. Please note that in our approach it is possible to model general aggregations, i.e., mixed aggregations of data and entity types.

Another important concept is association, sometimes called grouping [31] or cover aggregation. Given an object (entity) type o, an association over o is a type whose objects are sets of objects of type o. In this sense, the type TEAM would be an association over PERSON since each team (instance) consists of a set of members (of type PERSON). We could have modeled this fact by an entity type TEAM which has the component "members: TEAM → set(PERSON)" in our terminology. Thus, set-valued components completely support associations, while other ER approaches generally do not go beyond simple set-valued attributes.

On the other hand, some semantic data models extend the notion of association. For example, SDM provides expression-defined grouping, where the membership in one of the groups is defined by an expression, or enumerated grouping, where the members are explicitly itemized. We do not regard these extensions as basic primitives, but can express them by explicit integrity constraints [32].

Furthermore, specialization is a concept for modeling ISA or subset relationships. The simplest form consists of one subtype and one supertype; each instance of the subtype must be an instance of the supertype, too. This form corresponds to our notion of specialization, i.e., type constructions with one input (super) type and one output (sub) type.

Generally, several specializations $i_1, \ldots, i_n$ of the same supertype o are possible. However, they are independent of each other, so that an instance of the supertype occurs in more than one subtype. To model disjoint subtypes of a given supertype, a disjointness condition can be specified in SDM or IFO. We model this in our approach by using type constructions having one input type and several (disjoint) output types.

Yet another variant is offered by SDM in the form of set-operator defined subclasses, where the supertype is made up of subtypes by using the set operators "union," "intersection," or "difference." We can model the case "union of subtypes" with generalization. With respect to intersection or difference, we have once again to employ explicit integrity constraints.

On balance, we state that the concepts of aggregation, association, and specialization(s) in their general form seem to be integrated. In contrast to some data models, we strictly distinguish between modeling primitives and structural restrictions. Thus, some additional concepts such as expression-defined groupings (SDM) are not covered by the semantics, although they exist in the description of the EER model given by Hohenstein et al. [32]. From a semantic point of view , they can be defined formally by integrity constraints, or, roughly speaking, by formulas of the calculus presented in the following section.

This completes the foundation of our EER house. We are now ready to build the walls and the roof.

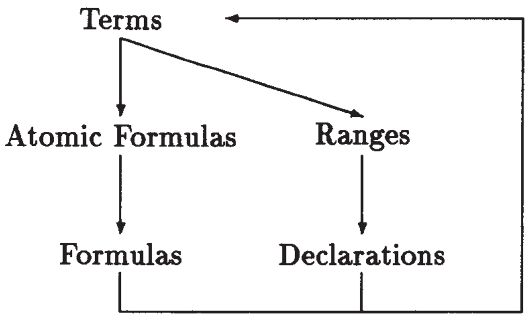
4. THE EER CALCULUS

Since the ER model is widespread and accepted, several high-level query languages based upon it are proposed, see for example [19–26]. But often, the definition of semantics does not go beyond an informal description by means of examples. To give a theoretical foundation, we want to present a precisely defined EER calculus, which can be used easily to fix the semantics of a query language by mapping the language into this calculus. But, as experience shows for the relational model, both relational tuple and domain calculus [37] are not powerful enough to define the semantics of relational query languages. Whereas relational languages like SQL or Quel provide features for data operations and aggregate functions, the calculi do not. Klug [34] has been the first to suggest an extended relational calculus including at least aggregate functions; whereupon Bültzingsloewen [7] has shown how to map SQL into (a variant of) this calculus.

We take up the idea of Klug [34], but we strictly distinguish between sets and bags, while he only uses sets and has to simulate the retention of duplicates. Moreover, we have laid the foundation for a formal definition of sets, bags, and aggregate functions in Section 3. Similar to Klug, we discard the usual hierarchical structure of the predicate calculus

terms
↓
atomic formulas
↓
formulas

and propose the nonhierarchical structure given by



(in both cases the arrow denotes "is-used-for").

This structure is necessary, because we define arbitrary bag-valued terms of the form

$$\{\!| t_1, \ldots, t_n \mid \delta_1 \wedge \cdots \wedge \delta_k \wedge \varphi |\!\}.$$

The terms $t_1, \ldots, t_n$ compute the target information, $\delta_1, \ldots, \delta_k$ are declarations of the form $(x : \rho)$ restricting the values for the variable x to the finite range ρ, and φ is a qualifying formula. The declarations δ_i can again use bag-valued terms of the above form as ranges. The introduction of ranges and declarations is indispensable to prevent infinite bags that lead to problems concerning aggregate functions.

Please note that the brackets $\{\!| \ldots |\!\}$ are the *syntactical* notation for bags in the calculus, in contrast to the *semantical* notation for bags $\{\{\ldots\}\}$ or sets $\{\ldots\}$ in the metalanguage. We only have arbitrary bags, but sets can be derived from bags by applying the function BTS.

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 ∪ ENTTYPE ∪ RELTYPE

We extend the notions of operations (OPNS) and predicates (PRED) introduced for data signatures to sort expressions and EER schemas resulting in the following notation:

$OPNS_{EER} :=$ ATTRIBUTE $\cup$ COMPONENT
$PRED_{EER} :=$ RELTYPE

All operations and predicates are now concentrated to

$OPNS := OPNS_{DS} \cup OPNS_{EER} \cup OPNS(SORT)$
$PRED := PRED_{DS} \cup PRED_{EER} \cup PRED(SORT)$

μ[SORT], $\mu[OPNS_{EER}]$, $\mu[PRED_{EER}]$, 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, ENTTYPE, and RELTYPE are disjoint as well as all (interpretations of) set-/list-/bag-constructed sort expressions (except for the special value $\perp$). $\hat{\mu}$[EXPR(SORT)] = $\hat{\mu}$[EXPR(DATA $\cup$ ENTTYPE $\cup$ RELTYPE)] is the union of all possible values found in (the interpretation of) DATA, ENTTYPE or RELTYPE or sort expressions built over these sets. Due to the disjointness there is a unique function

$$\text{sort}: \hat{\mu}[\text{EXPR(SORT)}] - \{\perp\} \to \text{EXPR(SORT)}$$

yielding, for an instance of $\hat{\mu}$[EXPR(SORT)], the sort it belongs to; the value $\perp$ belongs to every sort.

To formally define the syntax and semantics of the objects of our calculus we have to introduce variables and assignments substituting values for these variables.

Definition. (Variables and assignments). Let an EER schema EER(DS), a set VAR of *variables* and a function *type*: VAR $\to$ EXPR(SORT) be given ($v \in VAR_s$ stands for $v \in VAR$ and type(v) = s). The *set of assignments* ASSIGN is defined by

$$\text{ASSIGN} := \{\alpha \in |\text{FUN}| \mid \alpha: \text{VAR}_s \to \hat{\mu}[\text{EXPR(SORT)}](s) \text{ for every } s \in \text{EXPR(SORT)}\}$$

The special assignment ε: VAR $\to \{\perp\}$ is called the *empty assignment*.

Assignments are needed to handle the context for the evaluation (or semantical definition) of *terms*, *formulas*, *ranges*, and *declarations*. For example, a variable $v \in VAR$ is a correct term. The semantics of this term is then given by $\alpha(v)$, since terms must evaluate to values.

We now present the syntax and the semantics for each part of our calculus. As we will see from time to time, many simplifications can be introduced as "syntactic sugar," without, however, losing formal semantics. We assume one fixed EER schema EER(DS), and start with a preliminary definition of declarations for ease of demonstration. The set of all declarations is called DECL. Declarations are used to bind variables to finite sets. The set of declared (or bound) variables of a declaration δ is called decl(δ), the set of all free variables (i.e., variables, that come from "outside," but appear in the declaration) is called free(δ).

Definition. (Declarations, simple form). The *syntax of declarations* is given by a set DECL and functions *free*, *decl*: DECL $\to$ $\mathcal{F}$(VAR).

(i) If $v \in VAR_s$ for $s \in ENTTYPE$ or $s \in RELTYPE$, then $(v : s) \in DECL$, $free((v : s)) = \emptyset$, and $decl((v : s)) = \{v\}$.

The *semantics of declarations* is a relation $\mu[DECL] \subseteq DECL \times ASSIGN$.

(i) $((v : s), \alpha) \in \mu[DECL]$, *iff* $\alpha(v) \in \mu[SORT](s)$. (This property is already fulfilled by the definition of assignments.)

Please note that in declarations we only allow variables to range over (finitely interpreted) entity or relationship types, but not over (possibly infinitely interpreted) data types at this point. This is the simplest form of variable declaration, similar to the relational calculus defined by Pirotte [43]. Every used variable must be declared in this way at an appropriate place. By its declaration the variable becomes bound to a finite set of values. For later purposes, we also need a function *free*. Indeed, at this point there are no free variables in declarations.

Terms represent syntactical categories, which can be evaluated to yield a value of a certain sort. Therefore, we introduce the function sort: TERM $\rightarrow$ EXP(SORT). As in the case of declarations, terms can contain variables coming from "outside." These are the free variables of the term: free: TERM $\rightarrow$ F(VAR).

Definition. (Terms). The *syntax of terms* is given by a set *TERM*, and functions *sort*: TERM $\rightarrow$ EXPR(SORT) 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 $v \in VAR_r$, $r(e_1, \ldots, e_n) \in RELTYPE$ and $roles(r) = \langle n_1, \ldots, n_n \rangle$, then $v.i \in TERM_{e_i}$ ($i = 1..n$) and $v.n_i \in TERM_{e_i}$ ($i = 1..n$) with $free(v.i) := \{v\}$ and $free(v.n_i) := \{v\}$.
(iii) If $c \in CONSTRUCTION$, $s_{out} \in output(c)$ and $t_{in} \in TERM_{s_{in}}$, then $s_{out}(t_{in}) \in TERM_{s_{out}}$ with $free(s_{out}(t_{in})) := free(t_{in})$.
(iv) If $c \in CONSTRUCTION$, $s_{in} \in input(c)$, and $t_{out} \in TERM_{s_{out}}$, then $s_{in}(t_{out}) \in TERM_{s_{in}}$ with $free(s_{in}(t_{out})) := free(t_{out})$.
(v) If ω: $s_1 \times \cdots \times s_n \rightarrow s \in OPNS$ and $t_i \in TERMS_{s_i}$ ($i = 1..n$), then $\omega(t_1, \ldots, t_n) \in TERM_s$ with $free(\omega(t_1, \ldots, t_n)) := free(t_1) \cup \cdots \cup free(t_n)$.
(vi) If $t_i \in TERMS_{s_i}$ $(i = 1..n)$, $\delta_j \in DECL$ $(j = 1..k)$, $\varphi \in FORM$, and $decl(\delta_i) \cap decl(\delta_j) = \emptyset$ $(i \neq j)$, $decl(\delta_i) \cap free(\delta_j) = \emptyset$, then

$$\{\!\!\{ t_1, \ldots, t_n \mid \delta_1 \wedge \cdots \wedge \delta_k \wedge \varphi \}\!\!\} \in TERM_{bag(prod(s_1, \ldots, s_n))}$$

and $free(\{\!\!\{ t_1, \ldots, t_n \mid \delta_1 \wedge \cdots \wedge \delta_k \wedge \varphi \}\!\!\}) := (free(\delta_1) \cup \cdots \cup free(\delta_k) \cup free(t_1) \cup \cdots \cup free(t_n) \cup free(\varphi)) - (decl(\delta_1) \cup \cdots \cup decl(\delta_k))$.

The *semantics of terms* is a function $\mu[TERM]$: TERM $\times$ ASSIGN $\rightarrow$ $\hat{\mu}[EXPR(SORT)]$.

(i) $\mu[TERM](v, \alpha) := \alpha(v)$.
(ii) $\mu[TERM](v.i, \alpha) :=$ *if* $\mu[TERM](v, \alpha) = (x_1, \ldots, x_n)$ *then* x_i *else* $\perp$.

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

(iv) μ[Term]($s_{in}(t_{out})$, α):= *if* $\mu[\text{CONSTRUCTION}](c)(\mu[\text{TERM}](t_{out}, \alpha))$
$\in \mu[\text{ENTTYPE}](s_{in})$
then $\mu[\text{CONSTRUCTION}](c)(\mu[\text{TERM}](t_{out}, \alpha))$
else $\perp$.

(v) $\mu[\text{TERM}](\omega(t_1, \ldots, t_n), \alpha)$:=

$$\mu[\text{OPNS}](\omega)(\mu[\text{TERM}](t_1, \alpha), \ldots, \mu[\text{TERM}](t_n, \alpha)).$$

(vi) $\mu[\text{TERM}](\{t_1, \ldots, t_n \mid \delta_1 \wedge \cdots \wedge \delta_k \wedge \varphi\}, \alpha)$:=

$$\{\{(\mu[\text{TERM}](t_1, \alpha'), \ldots, \mu[\text{TERM}](t_n, \alpha')) \mid \text{there is } \alpha' \in \text{ASSIGN with } \alpha'(v) = \alpha(v) \text{ for } v \in \text{VAR} - (\text{decl}(\delta_1) \cup \cdots \cup \text{decl}(\delta_k)) \text{ and } (\delta_1, \alpha') \in \mu[\text{DECL}] \text{ and } \cdots \text{ and } (\delta_k, \alpha') \in \mu[\text{DECL}] \text{ and } (\varphi, \alpha') \in \mu[\text{FORM}]\}\}.$$

The variables $v \in \text{VAR}$ are the elementary form (i) of terms. Starting with them, we can build new terms by selecting relationship components (ii) and by applying operations (v). Remember that the operations $\omega \in \text{OPNS}$ may be data operations (from OPNS_{DS}), operations for sort expressions (from OPNS(SORT)) like MIN or SUM, or attributes and components (from OPNS_{EER}). Furthermore, we have the type conversions (iii) and (iv) respecting the semantics of type construction. If an entity e_{out} from entity sort s_{out} ($\in$ output(c)) is constructed from an entity e_{in} of sort s_{in} ($\in$ input(c)), i.e., $\mu[\text{CONSTRUCTION}](c)\ (e_{out}) = e_{in}$ holds, then we have the facts $s_{out}(e_{in}) = e_{out}$ and $s_{in}(e_{out}) = e_{in}$. (vi) allows the specification of arbitrary bag-valued terms presented at the beginning of this section.

Please note that $\mu[\text{TERM}](\{t_1, \ldots, t_n \mid \delta_1 \wedge \cdots \wedge \delta_k \wedge \varphi\}, \alpha)$ can evaluate to a proper bag, although the assignment α is fixed. The bag arises because the terms t_i are evaluated with respect to all assignments α', which can differ from α in the values of the variables declared in the declarations δ. Additionally, the assignments α' have to fulfill the declarations δ_i and the formula φ. Therefore, the (finite) number of the assignments α' is also a measure for the number of occurrences of an element in the resulting bag. For example, if we have a database state containing four people, then there are four different assignments α' in the evaluation of $\mu[\text{TERM}](\{\text{age}(p) \mid p: \text{PERSON}\}, \alpha)$, and the resulting bag could look like $\{\{25, 42, 42, 60\}\}$. If additionally there are two different countries in the state, then the evaluation of $\mu[\text{TERM}](\{\text{age}(p) \mid p: \text{PERSON} \wedge c: \text{COUNTRY}\}, \alpha)$ will yield a bag of integers with cardinality eight, i.e., $\{\{25, 25, 42, 42, 42, 42, 60, 60\}\}$. This is due to the fact that there are eight different assignments α'.

The function *sort* determines the sort of every term. The function *free* yields the set of free variables for a given term. The free variables are in fact the ones that have to be given a value by an assignment. The assignments to nonfree variables are not essential for the evaluation of terms. In cases (i)–(v) the free variables are obvious. In case (vi) all free variables of the terms $t_1, \ldots, t_n$, of the declarations $\delta_1, \ldots, \delta_k$ (at the moment there are no free variables in declarations; but see the later definition of declarations), and in the formula φ are free, except those declared in $\delta_1, \ldots, \delta_k$.

For simplicity, we assume a formula φ being omitted to be the tautology *true*. To handle constants c of data sorts d ($d \in DATA$) easily, constants are assumed to be 0-ary functions c: $\rightarrow$ d.

The semantics of terms is often called the evaluation of the terms with respect to a given assignment α of values to the variables.

Remark. (Variables used in the rest of this section). In the following examples we assume the variable r to be of type *RIVER*, s of type *SEA*, w of type *WATERs*, t of type *TOWN*, c of type *COUNTRY*, p of type *PERSON*, l of type *LAKE*, la of type *lies-at*, ft of type *flows-through*, a of type *address*, rg of type *region*, f of type *forms*, str of type *string*, and i of type *int*. Additionally, we use indices, i.e., s_1 and s_2 are variables of type *SEA*.

Examples. (Terms)

(i) (1) **c** is a term of sort *COUNTRY* having only one free variable c (of type *COUNTRY*). For a given assignment α, this term evaluates to the value assigned to the variable c by α, i.e., a specific country or the undefined value $\perp$.
la is a term of sort *lies-at* having only one free variable la. This term evaluates to the value assigned to the variable la by α, i.e., a specific relationship between the town *la.TOWN* and the river *la.RIVER* (see (ii)).

(ii) (2) **la.TOWN** and **la.1** are both terms of sort *TOWN* having the single free variable la. Both terms evaluate to the town that participates in the relationship assigned to the variable la by a given α.

(iii) (3) **WATERS(r)** is a term of sort *WATERS* having the single free variable *r*. This term evaluates for a given α to the body of water corresponding to the river assigned to r by α. Indeed, we assumed that every river is also a body of water (by an additional constraint).

(iv) (4) **RIVER(w)** is a term of sort *RIVER* having the single free variable w. This term evaluates to the river that corresponds to the body of water assigned to w by α. If this body of water is a sea or a lake, and not a river, the term evaluates to undefined ($\perp$).

(v) (5) **cname(c)** is a term of sort *string*, since c is a term of sort *COUNTRY* and *cname* a function cname: COUNTRY $\rightarrow$ *string*. This term evaluates to the name of the country assigned to the variable c by a given α.
distance(la) is a term of sort *real*, and evaluates for a given α to the distance of the town *la.TOWN* from the river *la.RIVER* for a specific relationship assigned to the variable la by α.
The variable c is free in cname(c) and the variable la is free in distance(la).

(6) **pname(boss(c))** is a term of sort *string*, since we have

$$\begin{array}{lll} \text{boss:} & \text{COUNTRY} \rightarrow \text{PERSON} & \\ \text{pname:} & & \text{PERSON} \rightarrow \textit{string}. \end{array}$$

and c is the only free variable. This term evaluates to the name of the head of the country assigned to c by a given α.

(7) **ministers(c)** is a term of sort *list*(*PERSON*) having the single free variable c.

(8) **CNT(ministers(c))** is a term of sort *int* having the single free variable c. This term evaluates to the number of ministers of the country assigned to c by a given α.

(9) Take care: **pname(ministers(c))** is not a correct term, since we have

$$\begin{array}{ll} \text{ministers: COUNTRY} \rightarrow \text{list(PERSON) and} & \\ \text{pname:} & \text{PERSON} \rightarrow \textit{string}\text{, but not} \\ \text{pname:} & \text{list(PERSON)} \rightarrow \text{list}(\textit{string})! \end{array}$$

However, this natural extension of functions σ: $s_1 \rightarrow s_2$ to functions σ: list(s_1) $\rightarrow$ list(s_2) could be introduced as "syntactic sugar." A correct formulation of the intended term is given later by example (13).

(10) **3.14*radius(tgeo(t))** is a term of sort *real*, since we have

$$\begin{array}{lll} \text{tgeo:} & \text{TOWN} \rightarrow \textit{circle}, & \\ \text{radius:} & \textit{circle} \rightarrow \textit{real}, & \\ \text{3.14:} & & \rightarrow \textit{real}, \\ \text{*:} & & \textit{real} \times \textit{real} \rightarrow \textit{real}. \end{array}$$

and t is the only free variable in this term.

(11) **pdist (centre(tgeo(t_1)), centre(tgeo(t_2)))** is a term of sort *real* having the free variables t_1 and t_2. It evaluates to the distance of the towns t_1 and t_2 for a given α.

(vi) (12) **{cname(c), pname(boss(c)) | (c : COUNTRY)}** is a term of sort *bag* (*prod*(*string*, *string*)) without free variables, since the variable c is bound by the declaration (c : *COUNTRY*). This term evaluates (independently of any α) to the names of all countries together with the names of the heads of those countries. Since the formula φ is missing, we assume φ to be *true*.

(13) **{pname(p) | (p : PERSON) $\wedge$ p IN ministers(c)}** is a correct term of sort *bag*(*string*) corresponding to the intended term in (9). (Indeed, if we want to be pedantic, we say this term has sort *bag*(*prod*(*string*)). But we do not distinguish between the sorts *prod*(s) and s, if s is a sort expression.) The target term *pname*(p) is a term of sort *string*; (p : *PERSON*) is a simple form of the declaration binding the variable p to the elements of PERSON (to be exact, to the elements of μ[ENTTYPE](PERSON)), and p *IN ministers*(c) is a simple formula containing the predicate *IN*.

The variable c is the only free variable in this term, since c is free in the formula and not declared by the declaration. Indeed, p is free in the target term, but becomes bound by the declaration to obtain sort *PERSON*.

For a given assignment α, this term evaluates to the names of all persons (in PERSON) that are minister of the country assigned to c by α.

(14) **-{cname(c) | (c : COUNTRY) ∧ (c : TOWN)}-** is not a correct term since the variable c is declared twice. This disagrees with the fact that the variable c has exactly one type (namely COUNTRY).

(15) **-{cname(c), [tname(t) | (t : TOWN) ∧ lies-in(c, t)] | (c : COUNTRY) ∧ government(c) = democratic}-** is a term of sort *bag(prod(string, bag(string)))* without free variables. It yields for each stored democratic country the bag of all towns in the country. We see that it is possible to have bags as target terms, allowing nested bag-valued terms.

General Remark. (Type construction and inheritance). There is a close connection between the notions of type construction and inheritance. Consider a type construction with input type s_{in} and output type s_{out}. If we have a term t_{in} of entity type s_{in}, we can also build the term $s_{out}(t_{in})$ of entity type s_{out} (assuming a_{out} is an attribute of entity type s_{out}), and the term $a_{out}(s_{out}(t_{in}))$ is well formed. Please note that this makes it possible to apply attribute a_{out} to a term of sort s_{in}, and, in a way, means that entity type s_{in} inherits attribute a_{out} via type construction. Analogously, we can do this for the other direction in the type construction and build terms of the form $a_{in}(s_{in}(t_{out}))$. Of course, there is a price to pay for this flexibility: (1) we must insist on the injectivity of the semantic functions μ[CONSTRUCTION](c) for each type construction c; and (2) our approach relies on the requirement that functions may yield the undefined value $\perp$. This is necessary for terms like rgeo(RIVER(w)) if the current "w" is not a river.

Formulas are syntactic categories representing truth values. As in the case of terms, they can have free variables coming from "outside."

Definition. (Formulas). The *syntax of formulas* is given by a set *FORM* and a function *free*: FORM → F(VAR).

(i) If $\pi: s_1 \times \cdots \times s_n \in$ PRED and $t_i \in \mathrm{TERM}_{s_i}$ (i = 1..n), then $\pi(t_1, \ldots, t_n) \in$ *FORM* and $\mathrm{free}(\pi(t_1, \ldots, t_n)) := \mathrm{free}(t_1) \cup \cdots \cup \mathrm{free}(t_n)$.

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

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

(iv) If $c \in$ CONSTRUCTION, $s_{in} \in$ input(c), $s_{out} \in$ output(c), $t_{in} \in \mathrm{TERM}_{s_{in}}$, and $t_{out} \in \mathrm{TERM}_{s_{out}}$, then t_{in} *IS* $t_{out} \in$ *FORM* with $\mathrm{free}(t_{in}\ \mathrm{IS}\ t_{out}) := \mathrm{free}(t_{in}) \cup \mathrm{free}(t_{out})$.

(v) If $\varphi \in$ FORM, then $\neg(\varphi) \in$ *FORM* and $\mathrm{free}(\neg(\varphi)) := \mathrm{free}(\varphi)$.

(vi) If $\varphi_1, \varphi_2 \in$ FORM, then $(\varphi_1 \vee \varphi_2) \in$ *FORM* and $\mathrm{free}((\varphi_1 \vee \varphi_2)) := \mathrm{free}(\varphi_1) \cup \mathrm{free}(\varphi_2)$.

(vii) If $\varphi \in \text{FORM}$ and $\delta \in \text{DECL}$, then $\exists\delta(\varphi) \in \mathit{FORM}$ and $\text{free}(\exists\delta(\varphi)) := (\text{free}(\varphi) - \text{decl}(\delta)) \cup \text{free}(\delta)$.

The *semantics of formulas* is a relation $\mu[\text{FORM}] \subseteq \text{FORM} \times \text{ASSIGN}$.

(i) $(\pi(t_1, \ldots, t_n), \alpha) \in \mu[\text{FORM}]$ *iff* $(\mu[\text{TERM}](t_1, \alpha), \ldots, \mu[\text{TERM}](t_n, \alpha)) \in \mu[\text{PRED}](\pi)$.

(ii) $(t_1 = t_2, \alpha) \in \mu[\text{FORM}]$ *iff* $\mu[\text{TERM}](t_1, \alpha) = \mu[\text{TERM}](t_2, \alpha)$.

(iii) $(\text{UNDEF}(t), \alpha) \in \mu[\text{FORM}]$ *iff* $\mu[\text{TERM}](t, \alpha) = \perp$.

(iv) $(t_{in}\ \text{IS}\ t_{out}, \alpha) \in \mu[\text{FORM}]$ *iff* $\mu[\text{CONSTRUCTION}](c)(\mu[\text{TERM}](t_{out}, \alpha)) = \mu[\text{TERM}](t_{in}, \alpha)$.

(v) $(\neg(\varphi), \alpha) \in \mu[\text{FORM}]$ *iff* not $(\varphi, \alpha) \in \mu[\text{FORM}]$.

(vi) $((\varphi_1 \vee \varphi_2), \alpha) \in \mu[\text{FORM}]$ *iff* $(\varphi_1, \alpha) \in \mu[\text{FORM}]$ or $(\varphi_2, \alpha) \in \mu[\text{FORM}]$.

(vii) $(\exists\delta(\varphi), \alpha) \in \mu[\text{FORM}]$ *iff* there is $\alpha' \in \text{ASSIGN}$ with $\alpha'(v) = \alpha(v)$ for $v \in \text{VAR} - \text{decl}(\delta)$ and $(\varphi, \alpha') \in \mu[\text{FORM}]$ and $(\delta, \alpha') \in \mu[\text{DECL}]$.

Formulas are defined in the conventional way: predicates (i) and equations between terms (ii) are usually known as *atomic formulas*. The test UNDEF for terms is necessary, since the special value $\perp$ is an element of every sort, but not a correct term (!). (v) and (vi) define the usual logical connectives *not* and *or*, whereas the *existential quantifier* is defined in (vii). The formula (iv) results from type construction. t_{in} *IS* t_{out} holds, iff t_{in} and t_{out} are terms representing the "same" entity, in the sense of type construction. The set of free variables is passed on in the definition of formulas; however, special attention must be dedicated to case (vii). Normally, every variable is left free that occurs free in the formula φ and is not declared in the declaration δ. This would result in

$$\text{free}_{\text{FORM}}(\exists\delta(\varphi)) = \text{free}_{\text{FORM}}(\varphi) - \text{decl}(\delta).$$

Because, later on, declarations can also have occurrences of free variables, we obtain the presented computation of free variables.

For simplicity, we define the usual logical connectives $\wedge$, $\Rightarrow$, $\Leftrightarrow$ by

$$(\varphi_1 \wedge \varphi_2) := \neg(\neg\varphi_1 \vee \neg\varphi_2),$$
$$(\varphi_1 \Rightarrow \varphi_2) := (\neg\varphi_1 \vee \varphi_2),$$
$$(\varphi_1 \Leftrightarrow \varphi_2) := (\varphi_1 \Rightarrow \varphi_2) \wedge (\varphi_2 \Rightarrow \varphi_1),$$

and the quantifier $\forall$ by

$$\forall\delta(\varphi) := \neg\exists\delta(\neg(\varphi)).$$

We assume that the operators $\forall/\exists$, $\neg$, $\wedge/\vee$, $\Rightarrow$, $\Leftrightarrow$ have descending priorities, so that we can omit unnecessary parentheses. Furthermore, we omit parentheses in case of

$$\omega(\{\!\!\{ \cdots \}\!\!\}) \text{ for } \omega \in \{\text{CNT}, \text{LTB}, \text{BTS}, \text{LTS}, \text{SEL}, \text{IND}, \text{POS}, \text{OCC}, \text{AVG}, \text{SUM}, \text{MIN}, \text{MAX}\} \cup \{\text{APL}_\sigma \mid \sigma \in \text{OPNS}^{\text{BIN}}\}.$$

Thus, we write $\omega\{\!\!\{ \cdots \}\!\!\}$ instead of $\omega(\{\!\!\{ \cdots \}\!\!\})$.

Examples. (Formulas)

(i) (1) **lies-at(t, r)** is a formula with the free variables t and r. This formula holds iff a given α assigns t to a town and r to a river such that the town lies at the river.

(2) **ppcut(sgeo(s_1), sgeo(s_2))** is a formula with free variables s_1 and s_2. This formula holds iff the polygons representing the seas assigned to s_1 and s_2 by α have points in common.

(3) **p IN ministers(c)** is a formula with free variables p and c. This formula holds iff the person assigned to p by α is a minister of the country assigned to c by α.

(ii) (4) **p** = **boss(c)** is a formula with free variables p and c. This formula holds iff the person assigned to p by α is the head of the country assigned to c by α.

(5) **LTB(ministers(c))** = **$\{\!\!\{$p | (p : PERSON) $\wedge$ age(p) > 50$\}\!\!\}$** is a formula with the only free variable c. This formula holds iff the bag of ministers of the country assigned to c by α is equal to the bag of persons that are more than 50 years old.

(iii) (6) **UNDEF(addr(p))** is a formula with free variable p. It holds iff the person assigned to p by α has an unknown address.

(iv) (7) **s IS w** is a formula with the free variables s and w. It holds iff the body of water assigned to w by α "is" the sea assigned to s.

(vii) (8) **$\forall$(t : TOWN) (tpopulation(t) > 100,000) $\Rightarrow$ $\exists$(p : PERSON) (is-mayor-of(p, t))** is a formula without free variables. This formula holds iff every town with more than 100,000 inhabitants has a mayor.

Up to now, there was no possibility of letting variables range over data types, since we allowed declarations of the form $(v : s)$, $s \in$ ENTTYPE or $s \in$ RELTYPE only. On the other hand, it makes no sense (for the formulation of queries) to allow declarations $(v : d)$ with $d \in$ DATA, because this leads to infinite ranges for variables, and it would then be possible to specify infinite bag-valued terms. To guarantee finite bags, we introduce the notion of *ranges* that restrict possibly infinite value sets for any variable to a finite set, i.e., a subset of actually "stored" values.

Definition. (Ranges). The *syntax of ranges* is given by a set *range* and functions *domain*: RANGE $\rightarrow$ EXPR(SORT) and *free*: RANGE $\rightarrow$ F(VAR) ($\rho \in \text{RANGE}_s$ stands for $\rho \in$ RANGE and domain(ρ) = s).

(i) If $s \in$ ENTTYPE or $s \in$ RELTYPE, then $s \in \mathit{RANGE}_s$ and $\text{free}_{\text{RANGE}}(s) := \emptyset$.

(ii) If $t \in \mathrm{TERM}_{set(s)}$ with $s \in \mathrm{EXPR(SORT)}$, then $t \in RANGE_s$ and $\mathrm{free}_{\mathrm{RANGE}}(t) := \mathrm{free}_{\mathrm{TERM}}(t)$.
(iii) If $\rho \in \mathrm{RANGE}_s$, then $set(\rho) \in RANGE_{set(s)}$ and $\mathrm{free}_{\mathrm{RANGE}}(\mathrm{set}(\rho)) := \mathrm{free}(\rho)$.

The *semantics of ranges* is a function $\mu[\mathrm{RANGE}]$: RANGE × ASSIGN → $\hat{\mu}[\mathrm{EXPR(SORT)}]$.

(i) $\mu[\mathrm{RANGE}](s, \alpha) := \mu[\mathrm{SORT}](s)$.
(ii) $\mu[\mathrm{RANGE}](t, \alpha) := \mu[\mathrm{TERM}](t, \alpha)$.
(iii) $\mu[\mathrm{RANGE}](\mathrm{set}(\rho), \alpha) := F(\mu[\mathrm{RANGE}](\rho, \alpha))$.

In addition to the ranges $s \in \mathrm{ENTTYPE}$ or $s \in \mathrm{RELTYPE}$ (already known from the simplified form of declarations), we can use any set-valued term as a range. As we will see, bag- or list-valued terms make no sense in this context.

Examples. (Ranges)

(i) (1) **PERSON** and **lies-at** are ranges with domain(PERSON) = PERSON and domain(lies-at) = *lies-at*. Both ranges do not have any free variables.
(ii) (2) **LTS(ministers(c))** is a range with domain *PERSON* having the single free variable c. The term **ministers(c)** is not a range, since it is list-valued.
(3) **area(c)** is a range of domain *region* having the single free variable c.
(iii) (4) **set(WATERS)** is a range with domain set(WATERS). $\mu[\mathrm{RANGE}](\mathrm{set(WATERS)}, \alpha)$ consists of all subsets of $\mu[\mathrm{RANGE}](\mathrm{WATERS})$. It could look like $\{\emptyset, \{w_1\}, \{w_2\}, \{w_3\}, \ldots, \{w_1, w_2\}, \{w_2, w_3\}, \ldots\}$. This form of range is used, for example, to compute the transitive closure (see general remark (1) that follows the examples for queries).

More examples for ranges will appear in the examples for declarations and queries.

Allowing variables to range over arbitrary ranges, we can now access data values stored in attributes of entities and relationships. But there is no possibility as yet to, for example, let a variable range over all addresses of all stored people, even if we use set-valued terms as ranges:

(*) (a: {a' | (a' : ?) ∧ (p : PERSON) ∧ a' IN addr(p)})

We remove this shortcoming by extending the form $(v : \rho)$ of declarations.

Definition. (Declarations, final form). The *syntax of declarations* is given by a set *DECL* and functions *free*, *decl*: DECL → F(VAR).

(i) If $v \in \mathrm{VAR}_s, \rho_1, \ldots, \rho_n \in \mathrm{RANGE}_s$, and not $v \in \mathrm{free}(\rho_1) \cup \cdots \cup \mathrm{free}(\rho_n)$, then $(v\colon \rho_1 \cup \cdots \cup \rho_n) \in DECL$, $\mathrm{free}(v\colon \rho_1 \cup \cdots \cup \rho_n) := \mathrm{free}(\rho_1) \cup \cdots \cup \mathrm{free}(\rho_n)$, and $\mathrm{decl}(v\colon \rho_1 \cup \cdots \cup \rho_n) := \{v\}$.
(ii) If $v \in \mathrm{VAR}_s$, $\rho_1, \ldots, \rho_n \in \mathrm{RANGE}_s$, $\delta \in \mathrm{DECL}$ with $\emptyset \neq (\mathrm{free}(\rho_1) \cup \cdots \cup \mathrm{free}(\rho_n)) \subseteq \mathrm{decl}(\delta) \cup \mathrm{free}(\delta)$, and not $v \in \mathrm{free}(\delta) \cup \mathrm{decl}(\delta)$, then $(v\colon \rho_1 \cup \cdots \cup \rho_n; \delta) \in DECL$, $\mathrm{free}(v\colon \rho_1 \cup \cdots \cup \rho_n; \delta) := \mathrm{free}(\delta)$, and $\mathrm{decl}(v\colon \rho_1 \cup \cdots \cup \rho_n; \delta) := \{v\} \cup \mathrm{decl}(\delta)$.

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

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

(ii) (v: $\rho_1 \cup \cdots \cup \rho_n; \delta, \alpha) \in \mu$[DECL] *iff* $(\alpha(v) \in \mu$[RANGE](ρ_1, α) or $\cdots$ or $\alpha(v) \in \mu$[RANGE]$(\rho_n, \alpha))$ and $(\delta, \alpha) \in \mu$[DECL].

At first, we do not only allow the form $(v: \rho)$, but also a union (i) $(v: \rho_1 \cup \cdots \cup \rho_k)$. This form is necessary to express every term of the relational algebra, especially the union of sets, in our calculus. In declarations of the form (ii) $(v_1 : \rho_1); \ldots ; (v_n : \rho_n)$, the variables $v_1, \ldots, v_n$ (in front of the colons) are declared (each $(v_i : \rho_i)$ 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 $\{v_{i+1}, \ldots, v_n\}$, plus the free variables of ρ_n. Indeed, the free variables of ρ_n are the free variables of the declaration, which must be disjoint from the variables declared in the declaration: free$(\delta) \cap$ decl$(\delta) = \emptyset$ holds for all declarations δ. 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.

With this sequence of assignments we are able to refer to the values of any data type stored in the database. We can now solve the problem (*) given above, by simply writing

(*) (a : LTS(addr(p))); (p : PERSON)

The variable a is bound to all addresses of all stored people. Please note that the variable p is declared, too.

Since the semantics of a declaration is a special form of formula using the element-of relation, it is not wise to allow list- or bag-valued terms as ranges. Whether we ask for $\alpha(v) \in \mu$[RANGE](ρ, α) for sets ρ or bags and lists ρ, the effect will be the same.

Examples. (Declarations)

(i) (1) **(p : PERSON)** is a declaration of variable p (decl(p : PERSON) = {p}), binding the variable p to persons, i.e., α assigns to the variable p a single value out of μ[ENTTYPE](PERSON). This declaration has no free variables.

(2) **(a : LTS(addr(p)))** is a declaration of variable a with the single free variable p, which binds a to the addresses of the person assigned to the variable p by a given α.

(3) **(p : LTS(ministers(c)))** is a declaration of variable p with the single free variable c, which binds p to the set of ministers of the country assigned to c by α.

(4) **(str : BTS-{tname(t) | (t : TOWN)}- ∪ BTS-{city(a) | (a : LTS(addr(p))); (p : PERSON)}-)** is a declaration of the variable str without free variables. The variable str is bound to the union of all stored town and city names appearing in the addresses of people.

(5) **(f : government(c))** is not a correct declaration, since government(c) is not a set-valued term, and consequently not a correct range. The intended formulation is the following declaration.

(6) **(f : BTS-{government(c) | (c : COUNTRY)}-)** is a declaration of variable f without free variables. The variable f is bound (independently of any assignment) to the forms of government (of each country).

(ii) (7) **(a : LTS(addr(p))); (p : LTS(ministers(c)))** is a declaration of the variables a and p with the single free variable c. The variable p is bound to the set of ministers of the country assigned to c by a given α, and the variable a is bound to the addresses of the ministers assigned to p by α. Thus, to satisfy this declaration, the assignment α, which already assigns a country to c, must assign to the variable p a minister of the country (from μ[RANGE](LTS(ministers(c)), α)) and to the variable a an address of the minister (from μ[RANGE](LTS(addr(p)), α)); roughly speaking, variable a ranges over the addresses of all ministers of the country assigned to c by α. Note that two declarations such as (a : LTS(addr(p))) ∧ (p : LTS(ministers(c))) are forbidded as part of a term (due to point (vi) of the term definition), because p is free in the left half and becomes declared in the right half.

(8) **(t : BTS-{t | (t : TOWN) ∧ is-mayor-of(p, t)}-); (p : LTS(ministers(c)))** is a declaration of the variables t and p having the single free variable c. The variable t is bound to the towns having a minister of the country (assigned to c by a given α) as mayor. Note that there is no conflict between the variable t declared in the bag and in the outer declaration.

(9) **(p : LTS(ministers(c))); (rg : area(c)); (c : COUNTRY)** is a correct declaration of the variables p, rg, and c without free variables. The variable c ranges over countries, the variable rg ranges over the regions of the countries, and the variable p ranges over the ministers of the countries.

To complete our EER calculus, we have to define *queries*. In principle, every term that computes a value, a set, bag, or list and does not have free variables can be considered as a query. However, we restrict queries to ensure that their results are "printable." Thus, as queries, we allow multivalued terms that have a sort constructed using data types only.

Definition. (Queries). The *syntax of queries* is given by a set QUERY ∈ |SET|. If $t \in \mathrm{TERM}_s$ with $s \in \mathrm{EXPR(DATA)}$ and free(t) = ∅, then $t \in$ *QUERY*.

The *semantics of queries* is given by a function μ[QUERY]: QUERY → μ[EXPR(DATA)]: μ[QUERY](t) := μ[TERM](t, ε).

Examples. (Queries)

(1) "Names of italian rivers"

$\{$rname(r) | (r : RIVER) $\wedge$ $\exists$(c : COUNTRY) (cname(c) = 'Italy' $\wedge$ flows-through(c, r))$\}$

or

$\{$rname(r) | (r : RIVER) $\wedge$ $\exists$(ft : flows-through) (cname(ft.1) = 'Italy' $\wedge$ ft.2 = r)$\}$

or

$\{$rname(ft.RIVER) | (ft : flows-through) $\wedge$ cname(ft.COUNTRY) = 'Italy'$\}$

(2) "Names of towns lying at most 10 km from a river"

$\{$tname(t) | (t : TOWN) $\wedge$ $\exists$(la : lies-at) (la.TOWN = t $\wedge$ distance(la) $\leq$ 10,000)$\}$

or

$\{$tname(la.TOWN) | (la : lies-at) $\wedge$ distance(la) $\leq$ 10,000$\}$

(3) "Names of rivers that flow through all socialistic countries"

$\{$rname(r) | (r : RIVER) $\wedge$
$\forall$(c : COUNTRY) (government(c) = socialistic $\Rightarrow$
flows-through(c, r))$\}$

(4) "Names of rivers that flow through France and their lengths in France"

$\{$rname(r), length(ft) | (r : RIVER) $\wedge$ (ft : flows-through) $\wedge$
$\exists$(c : COUNTRY) (cname(c) = 'FRANCE' $\wedge$ ft.RIVER
= r $\wedge$ ft.COUNTRY = c)$\}$

or

$\{$rname(ft.RIVER), length(ft) | (ft : flows-through) $\wedge$ cname(ft.COUNTRY) = 'France'$\}$

(5) "Names of countries having towns with less than 100,000 inhabitants only"

$\{$cname(c) | (c : COUNTRY) $\wedge$
$\neg\exists$(t : TOWN) (tpopulation(t) $\geq$ 100,000 $\wedge$ lies-in(c, t))$\}$

or

$\{$cname(c) | (c : COUNTRY) $\wedge$
$\forall$(t : TOWN) (tpopulation(t) $\geq$ 100,000 $\Rightarrow$ $\neg$lies-in(c, t))$\}$

or

$\{$cname(c) | (c : COUNTRY) $\wedge$
$\forall$(t : TOWN) (lies-in(c, t) $\Rightarrow$ tpopulation(t) < 100,000)$\}$

(6) "Names and areas of towns lying at the Rhine and having an area greater than 5 km^2"

$\{$tname(t), 3.14*radius(tgeo(t))↑2 | (t : TOWN) $\wedge$
$\exists$(r : RIVER) (rname(r) = 'Rhine' $\wedge$ lies-at(r, t) $\wedge$
3.14*radius(tgeo(t))↑2 > 5,000,000)$\}$

(7) "Names of the regions of the United Kingdom"

{name(area(c)) | (c : COUNTRY) ∧ cname(c) = 'United Kingdom'}

is not a correct query,

since area: COUNTRY → *regions*[= set(*region*)] and
name: *region* → *string*.

{name(rg) | (rg : area(c)); (c : COUNTRY) ∧ cname(c) = 'United Kingdom'}

is a correct formulation of this query using form (ii) of declarations. This form enables us to affect the result of a query in a way similar to the *nest/unnest* operators of the NF^2 model [33], as shown by the next examples.

(8) "Addresses of the ministers of democratic countries"

(a) {LTB(addr(*p*)) | (*p* : LTS(ministers(*c*))); (*c* : COUNTRY) ∧ government(*c*) = democratic}

yields the bag of addresses for each minister of a democratic country:

$$\left\{\left\{ \underbrace{\{\{\text{address}_{1,1}, \ldots, \text{address}_{1,k_1}\}\}}_{(\text{addresses of minister}_1)}, \ldots, \underbrace{\{\{\text{address}_{n,1}, \ldots, \text{address}_{n,k_n}\}\}}_{(\text{addresses of minister}_n)} \right\}\right\}$$

Please note that if there is a minister with ministries of two different countries, then his or her bag of addresses appears twice.

(b) {a | (a : LTS(addr(p))); (p : LTS(ministers(c))); (c : COUNTRY) ∧ government(c) = democratic}

yields every address of every minister of all democratic countries:

$$\left\{\left\{ \underbrace{\text{address}_{1,1}, \ldots, \text{address}_{1,k_1}}_{(\text{addresses of minister}_1)}, \ldots, \underbrace{\text{address}_{n,1}, \ldots, \text{address}_{n,k_n}}_{(\text{addresses of minister}_n)} \right\}\right\}$$

(c) {{a | (a : LTS(addr(p)))} | (p : LTS(ministers(c))); (c : COUNTRY) ∧ government(c) = democratic}

has nearly the same result as (a), but duplicate addresses (if there are any in the list of addresses for a single minister) are removed.

(d) {{addr(*p*) | (*p* : LTS(ministers(*c*)))} | (c : COUNTRY) ∧ government(c) = democratic}

yields, for all democratic countries, a bag containing for each minister of a country the list of his or her addresses:

$$\left\{\left\{ \underbrace{\{\{\text{addresslist}_{1,1}, \ldots, \text{addresslist}_{1,k_1}\}\}}_{(\text{democratic country}_1)}, \ldots, \underbrace{\{\{\text{addresslist}_{n,1}, \ldots, \text{addresslist}_{n,k_m}\}\}}_{(\text{democratic country}_m)} \right\}\right\}$$

with $\text{addresslist}_{i,j} = \langle \text{address}_1, \ldots, \text{address}_{m_{i,j}} \rangle$

$= \text{list of addresses of minister}_{i,j} \text{ belonging to country i}$

(9) "Names of ministers having addresses in the towns of their country only"

$$\{\text{pname}(p) \mid (p : \text{LTS}(\text{ministers}(c))); (c : \text{COUNTRY}) \wedge \forall (a : \text{LTS}(\text{addr}(p))) \exists (t : \text{TOWN}) (\text{lies-in}(c, t) \wedge \text{tname}(t) = \text{city}(a))\}$$

(10) "Names of rivers that flow into the North Sea"

$$\{\text{rname(r)} \mid (\text{r} : \text{RIVER}) \wedge \exists (\text{s} : \text{SEA}) (\text{sname(s)} = \text{'North Sea'} \wedge \exists (\text{w} : \text{WATERS}) (\text{s IS w} \wedge \text{flows-into(r, w)}))\}$$

We can also formulate this query using coercions:

$$\{\text{rname}(r) \mid (r : \text{RIVER}) \wedge \exists (s : \text{SEA}) (\text{sname}(s) = \text{'North Sea'} \wedge \text{flows-into}(r, \text{WATERS}(s)))\}$$

(11) "Names of bodies of water that have common points with Switzerland"

$$\{\text{wname(w)} \mid (\text{w} : \text{WATERS}) \wedge \exists (\text{rg} : \text{area(c)}); (\text{c} : \text{COUNTRY})(\text{cname(c)} = \text{'Switzerland'} \wedge (\text{ppcut (sgeo(SEA(w)), geo(rg))} \vee \text{lpcut (rgeo(RIVER(w)), geo(rg))} \vee \text{cpcut (lgeo(LAKE(w)), geo(rg))}))\}$$

(12) "Name and age of the 'boss' of the USSR"

$$\{\text{pname(p), age(p)} \mid (\text{p} : \text{PERSON}) \wedge \exists (\text{c} : \text{COUNTRY}) (\text{cname(c)} = \text{'USSR'} \wedge \text{p} = \text{boss(c)})\}$$

or

$$\{\text{pname(boss(c)), age(boss(c))} \mid (\text{c} : \text{COUNTRY}) \wedge \text{cname(c)} = \text{'USSR'}\}$$

(13) "Names of persons that have an unknown address"

$$\{\text{pname(p)} \mid (\text{p} : \text{PERSON}) \wedge \text{UNDEF(addr(p))}\}$$

(14) "Number of stored countries"

$$\text{CNT}\{\text{c} \mid (c : \text{COUNTRY})\}$$

(15) "Sum of inhabitants of all stored countries"

$$\text{SUM}\{\text{cpopulation}(c) \mid (\text{c} : \text{COUNTRY})\}$$

(16) "Average of country populations"

$$\begin{aligned}&\text{AVG}\{\text{cpopulation(c)} \mid (\text{c} : \text{COUNTRY})\} \equiv \\ &\text{SUM}\{\text{cpopulation(c)} \mid (\text{c} : \text{COUNTRY})\} /_r \\ &\text{CNT}\{\text{cpopulation(c)} \mid (\text{c} : \text{COUNTRY})\} \equiv \\ &\text{APL}_{+, \text{bag}(int)}\{\text{cpopulation}(c) \mid (c : \text{COUNTRY})\} /_r \\ &\qquad \text{CNT}_{\text{bag}(int)}\{\text{cpopulation(c)} \mid (\text{c} : \text{COUNTRY})\}\end{aligned}$$

(17) In the second section we formulated the query "Give me, for each stored country, its name and the average age of its ministers" as

$$\{\text{cname}(c), \text{AVG}\{\text{age}(p) \mid (p\text{:PERSON}) \wedge p \text{ IN ministers}(c)\} \mid (c\text{:COUNTRY})\}.$$

If we were pedantic, we would have to formulate the meaning of this calculus expression as "Give me for each stored country its name and the average age of the *people who are ministers* in this country," because a minister with two ministries is considered only once. If we really want "the average age of the ministers," taking into account the age of one person as often as she/he has ministries, we formulate this query as

$$\{\text{cname}(c),\ \text{AVG}\{\text{age}(p) \mid (i : \text{POS}(\text{ministers}(c), p));\ (p : \text{PERSON}) \land p \text{ IN } \text{ministers}(c)\} \mid (c : \text{COUNTRY})\}.$$

(18) "For every country its name and the average population of towns lying in it"

$$\{\text{cname}(c),\ \text{AVG}\{\text{tpopulation}(t) \mid (t : \text{TOWN}) \land \text{lies-in}(c, t)\} \mid (c : \text{COUNTRY})\}$$

This query is different from

$$\{\text{cname}(c),\ \text{AVG}\{\text{tpopulation}(t) \mid \text{lies-in}(c, t)\} \mid (c : \text{COUNTRY}) \land (t : \text{TOWN})\}$$

It yields for every country and for every town the name of the country and the average population of the town (i.e., the population of the town!), if the town lies in the country, and undefined otherwise.

(19) "Compute the average of the maximal populations of countries with a certain form of government over all these forms"

$$\text{AVG}\{\text{MAX}\{\text{cpopulation}(c) \mid (c : \text{COUNTRY}) \land \text{government}(c) = f\} \mid (f : \text{BTS}\{\text{government}(c) \mid (c : \text{COUNTRY})\})\}$$

Note: The result of this query is, in general, different from

$$\text{AVG}\{\text{MAX}\{\text{cpopulation}(c_1) \mid (c_1 : \text{COUNTRY}) \land \text{government}(c_1) = \text{government}(c_2)\} \mid (c_2 : \text{COUNTRY})\}$$

Assume we have three countries. The democratic country c_1 has 10,000,000 inhabitants, the democratic country c_2 has 20,000,000 inhabitants, and the socialistic country c_3 has 35,000,000 inhabitants. The first query computes for every form of government $f \in \{\text{democratic, socialistic}\}$ the maximal population of countries having this form, i.e., 20,000,000 for f = democratic and 35,000,000 for f = socialistic, and then the average of both values results in 27,500,000. The second query computes for each stored country c_1, c_2, and c_3 the maximal population of countries having the same form of government, i.e., 20,000,000 for c_1, 20,000,000 for c_2, and 35,000,000 for c_3. The average of the three is then 25,000,000.

(20) "Name and population of every country that has a population equal to the sum of the populations of its towns"

$$\{\text{cname}(c),\ \text{cpopulation}(c) \mid (c : \text{COUNTRY}) \land \text{cpopulation}(c) = \text{SUM}\{\text{tpopulation}(t) \mid (t : \text{TOWN}) \land \text{lies-in}(c, t)\}\}$$

(21) $\{\text{cname}(c),\ \text{AVG}\{\text{tpopulation}(t) \mid \text{lies-in}(c, t)\} \mid (c : \text{COUNTRY})\}$

is not a correct query, since the variable t in the inner bag is not declared in the outer bag: the query has a free variable.

(22) $\{p \mid (p : \mathrm{LTS}(\mathrm{ministers}(c))); (c : \mathrm{COUNTRY})\}$

is not a correct query, since the target term is not built from data sorts. The intended formulation of this query is probably

$$\{\mathrm{pname}(p), \mathrm{addr}(p), \mathrm{age}(p) \mid (p : \mathrm{LTS}(\mathrm{ministers}(c))); (c : \mathrm{COUNTRY})\}$$

Indeed, this could be introduced as "syntactical sugar."

(23)
$$\begin{array}{l}\{\mathrm{cname}(c),\\ \{\mathrm{tname}(t) \mid (t : \mathrm{TOWN}) \wedge \text{lies-in}(c, t)\},\\ \mathrm{cpopulation}(c) - (\mathrm{SUM}\{\mathrm{tpopulation}(t) \mid (t : \mathrm{TOWN}) \wedge \text{lies-in}(c, t)\}) \mid\\ (c : \mathrm{COUNTRY})\}\end{array}$$

yields for each stored country its name, the bag of all towns lying in it, and the difference between the country population and the sum of the populations of its towns.

(24)
$$\begin{array}{l}\{\mathrm{pname}(\mathrm{SEL}(\mathrm{ministers}(c), i)), i \mid (i : \mathrm{IND}(\mathrm{ministers}(c)))\\ (c : \mathrm{COUNTRY}) \wedge \mathrm{cname}(c) = \text{'USSR'}\}\end{array}$$

or

$$\begin{array}{l}\{\mathrm{pname}(p), i \mid (i : \mathrm{POS}(\mathrm{minister}(c), p)); (p : \mathrm{LTS}(\mathrm{minister}(c)));\\ (c : \mathrm{COUNTRY}) \wedge \mathrm{cname}(c) = \text{'USSR'}\}\end{array}$$

gives the names of all ministers of the USSR, together with a number indicating their position in the list of ministers. Please note that a minister's name can occur twice (with different position numbers).

(25)
$$\begin{array}{l}\{\mathrm{pname}(p) \mid (p : \mathrm{LTS}(\mathrm{ministers}(c))); (c : \mathrm{COUNTRY}) \wedge\\ \qquad \mathrm{CNT}\ (\mathrm{POS}(\mathrm{ministers}(c), p)) \geq 2 \wedge \mathrm{cname}(c) = \text{'USSR'}\}\end{array}$$

or

$$\begin{array}{l}\{\mathrm{pname}(p) \mid (p : \mathrm{LTS}(\mathrm{ministers}(c))); (c : \mathrm{COUNTRY}) \wedge\\ \qquad \mathrm{OCC}(\mathrm{LTB}(\mathrm{ministers}(c)), p) \geq 2 \wedge \mathrm{cname}(c) = \text{'USSR'}\}\end{array}$$

gives the names of all ministers of the USSR who have at least two ministries.

General Remarks.

(1) It is possible in our calculus to compute the transitive closure. For example, the following query computes the set of all bodies of water into which a river with a given name 'x' flows directly or indirectly.

$$\begin{array}{l}\{\mathrm{wname}(w) \mid (w : \mathrm{WATERS}) \wedge (ws : \mathrm{set}(\mathrm{WATERS})) \wedge w\ \mathrm{IN}\ ws \wedge\\ \qquad \forall (w_1 : \mathrm{WATERS})\\ \qquad\quad (w_1\ \mathrm{IN}\ ws \Leftrightarrow\\ \qquad\qquad \exists (r : \mathrm{RIVER})\ (\text{flows-into}(r, w_1) \wedge \mathrm{rname}(r) = \text{'x'})\\ \qquad\qquad \vee\\ \qquad\qquad \exists (w_2 : \mathrm{WATERS})\ (w_2\ \mathrm{IN}\ ws \wedge \text{flows-into}(\mathrm{RIVER}(w_2), w_1))\}\end{array}$$

Please note that the universally quantified formula has one free variable, namely, ws. The formula characterizes membership in ws.

(2) If we assume a total complete order $\leq$ on all data types, a query q of sort bag(prod($s_1, \ldots, s_n$)) can be converted into a term of sort list(prod ($s_1, \ldots, s_n$)) by specifying an order $\langle j_1, \ldots, j_n \rangle \in$ PERMUTATION(1, ..., n). This has to be done for inner bag-valued result terms as well. By doing this, the calculus can be enriched by sorting features according to a lexicographical order. For example, query (23) can be ordered by

$$\{\text{cname}(c), \{\text{tname}(t) \mid \cdots\}_{\text{order}:\langle 1\rangle}, \text{cpopulation}(c) - (\cdots) \mid \cdots\}_{\text{order}:\langle 2,1,3\rangle}$$

If we assume Bordeaux is the lexicographically first stored town in France, and that Berlin is the equivalent in Germany, the result could look like

⟨('Germany', ⟨'Berlin', ..., 'Neustadt', 'Neustadt', ...⟩, 100,000),
('France', ⟨'Bordeaux', ..., 'Paris', ...⟩, 75,000), ...⟩

We now prove two essential properties of our calculus, namely, safeness and relational completeness. If we again compare our job with that of building a house, we have now finished the foundation and the walls. The following properties represent the roof of our EER house.

THEOREM 1. *Every multivalued term $t \in TERM$ is evaluated to a finite set, bag, or list for every assignment α.*

PROOF. We prove by induction on the depth d of term construction that every term evaluates either to a single value or to a finite set, bag, or list for every assignment α.

Basis. (d = 1). The terms with depth 1 are variables v, i.e., terms of the form (i) or constants c. Indeed, these terms evaluate to the single value $\alpha(v)$ or to μ[OPNS](c) ( ).

Induction. Assume the proposition holds for any term t having a depth less than d + 1. Every term t′ with depth d + 1 can be of one of the following forms (ii)–(vi) only.

(ii) $t' \equiv \mathbf{t.i/t.n_i}$: Since the term t must have sort r, r ∈ RELTYPE, t can only be a variable with type r. The term t′ then evaluates to the i-th component of μ[TERM](t, α) ∈ μ[RELTYPE](r), i.e., a single entity (or undefined) for a given α.

(iii) $t' \equiv \mathbf{s_{out}(t_{in})}$: The term t′ evaluates for a given α to a single entity "being the same as" μ[TERM](t_{in}, α) in the sense of type construction. If such an entity does not exist, the term t′ evaluates to undefined.

(iv) $t' \equiv \mathbf{s_{in}(t_{out})}$: Analogous to (iii).

(v) $t' \equiv \boldsymbol{\omega}(\mathbf{t_1, \ldots, t_n})$: The term t′ evaluates either to a single value or to a set, bag, or list. In the latter case, μ[TERM](t′, α) must be finite because data operations ($\omega \in OPNS_{DATA}$), (multivalued) attributes and components ($\omega \in OPNS_{EER}$), and operations of sort expressions ($\omega \in OPNS_{EXPR}$) yield a finite result for finite arguments μ[TERM](t_i, α) (i = 1..n). Finiteness of μ[TERM](t_i, α) is guaranteed by the inductive assumption.

(vi) $t' \equiv \{\mathbf{t_1, \ldots, t_n} \mid \boldsymbol{\delta_1} \wedge \cdots \wedge \boldsymbol{\delta_k} \wedge \boldsymbol{\varphi}\}$: Each declaration δ_i (i = 1..k, k ≥ 0) is of the form $\delta_i \equiv \delta_{i,p_i}(v_{i,p_i}); \ldots; \delta_{i,1}(v_{i,1})$, and each $\delta_{i,j}$ (j = 1..p_i, $p_i \geq 1$) of the form $\delta_{i,j} \equiv (v_{i,j} : \rho^1_{i,j} \cup \cdots \cup \rho^q_{i,j})$ with $q \geq 1$, where the ρ's are ranges and the v's are variables. Speaking informally, the term t′ evaluates to a bag μ[TERM](t′, α) containing a tuple of the values μ[TERM](t_i, α') (i = 1..n) for every assignment α' that is equal to α except for the variables $v_{i,j}$ (declared in the δ's) and fulfills the delarations $\delta_1, \ldots, \delta_k$ as well as the formula φ. The terms t_i are at most of depth d. The ranges $\rho_{i,j}$ are either atomic (i.e., an entity or a relationship type), or of the form set(ρ), with ρ a range or set-valued terms having at most depth d.

By the inductive assumption, μ[TERM](t_i, α') is either a single value or a finite set, bag, or list. Thus, it is sufficient to prove that there is only a finite number of assignments α' satisfying the declarations δ_i.

If $\rho_{i,j}$ is an entity or relationship type s, μ[RANGE]($\rho_{i,j}$, α)) = μ[SORT](s) evaluates to a finite set. If $\rho_{i,j}$ is a set-valued term, $\rho_{i,j}$ must evaluate to a finite set μ[TERM]($\rho_{i,j}$, α) by induction. If $\rho_{i,j}$ is of the form set(ρ) with ρ atomic or set-valued, the range ρ evaluates to a finite set M and $\mathcal{F}$(M) is again finite. This argument extends to the general case, set(set(. . . ρ . . .)). Thus, each range $\rho_{i,j}$ evaluates to a finite set, and consequently each variable $v_{i,j}$ is bound to a finite set of values by the finite union $(v_{i,j} : \rho^1_{i,j} \cup \cdots \cup \rho^q_{i,j})$. Consequently, there is only a finite number of assignments α' assigning to a value $v_{i,1}$. For each of the values assigned to $v_{i,1}$, we obtain for the same reason a finite set of values for $v_{i,2}$, and so on.

Therefore, each variable $v_{i,j}$ appearing in the declarations δ_i (i = 1..k) is bound to a finite set of values, allowing a finite number of assignments α' only. ☐

LEMMA. *Since a query is a special case of a term, any query yields a finite result: the EER calculus is safe.*

Remark. There is no need to define *safe* queries in a semantical way, as for the relational calculi. Our approach results in a much cleaner expression of queries and removes the burden of checking for safe queries.

We now show that our calculus is relationally complete. This seems necessary to us, because relational completeness is a classical property of query languages. Nevertheless, in the light of the expressiveness of our calculus, this is a rather weak property.

THEOREM 2. *The EER calculus is relationally complete (if every relation is modeled by an entity type).*

PROOF. We prove the proposition by reducing the relational algebra to our calculus. The proof proceeds by induction on the number j of operators in an expression E of the relational algebra.

Basis. (j = 0, no operators). Then E is either a constant or a relation.

(i) Every constant c of a data sort d can be expressed by BTS{c | }
(ii) Every relation R with attributes $A_1, \ldots, A_n$ can be expressed by BTS{A_1(r), . . . , A_n(r) | (r : R)}

Please note that queries in the relational algebra result in a set of tuples.

Thus, we have to apply the converting function BTS to the constructed queries.

Induction. Assume the proposition holds for any relational expression E with fewer than j + 1 operators. Let E_i (i = 1, 2) have maximally j operators. By the inductive assumption, each E_i can be reduced to the queries

$$\rho_i \equiv \mathrm{BTS}\{ t_1^i, \ldots, t_{n_i}^i \mid \rho_1^i \wedge \cdots \wedge \rho_{k_i}^i \wedge \varphi^i \}.$$

Precisely these queries are set-valued terms, i.e., ranges. We obtain for every operator of the algebra the following queries in the calculus.

(i) $\mathbf{E_1 \cup E_2}$ **(union,** only significant if $n_1 = n_2$): $\mathrm{BTS}\{v.1, \ldots, v.n_1 \mid (v : \rho_1 \cup \rho_2)\}$

(ii) $\mathbf{E_1 - E_2}$ **(difference,** only significant if $n_1 = n_2$):

$$\mathrm{BTS}\{v_1.1, \ldots, v_1.n_1 \mid (v_1 : \rho_1) \wedge \neg\exists(v_2 : \rho_2)$$
$$(v_2.1 = v_1.1 \wedge \cdots \wedge v_2.n_2 = v_1.n_1)\}$$

(iii) $\mathbf{E_1 \times E_2}$ **(cartesian product):**

$$\mathrm{BTS}\{v_1.1, \ldots, v_1.n_1, v_2.1, \ldots, v_2.n_2 \mid (v_1 : \rho_1) \wedge (v_2 : \rho_2)\}$$

(iv) $\boldsymbol{\pi}_{\mathbf{l_1, \ldots, l_m}}(\mathbf{E_1})$ **(projection** on $l_1, \ldots, l_m \in 1..n_1$):

$$\mathrm{BTS}\{v.l_1, \ldots, v.l_m \mid (v : \rho_1)\}$$

(v) $\boldsymbol{\sigma}_{\mathbf{F}}(\mathbf{E_1})$ **(selection** with formulas $F \equiv (i = j)$ or $F \equiv (i = c)$, $i, j \in 1..n_1$, c constant): $\mathrm{BTS}\{v.1, \ldots, v.n_1 \mid (v : \rho_1) \wedge v.i = v.j\}$ or

$$\mathrm{BTS}\{v.1, \ldots, v.n_1 \mid (v : \rho_1) \wedge v.i = c\},$$

respectively.

These queries have the same result as the relational expressions. □

In the above proof, we do not mention the well-known join operation explicitly, since it is a derived operation employing selection first and projection afterwards. Nevertheless, in some sense, joins are inherent in the calculus because of the existence of relationship variables: if we have entity types e_1 with attributes $a_1, \ldots, a_n$ and e_2 with attributes $b_1, \ldots, b_k$ together with a binary relationship $r(e_1, e_2)$, we could define a *relationship join* of e_1 and e_2 via r in the following way:

$$\{a_1(v_r.e_1), \ldots, a_k(v_r.e_1), b_1(v_r.e_2), \ldots, b_k(v_r.e_2) \mid v_r : r\}$$

Thus, we collect all attribute values of two entities that are related with respect to a given relationship. But we not only know these relationship joins, but also *value-dependent joins*, taking into account properties of entities:

$$\{a_1(v_1), \ldots, a_k(v_1), b_1(v_2), \ldots, b_k(v_2) \mid v_1 : e_1 \wedge v_2 : e_2 \wedge a_i(v_1) \Theta b_j(v_2)\}$$

We assume attributes a_i and b_j have the same data type and Θ is an appropriate comparison operator.

Examples. (Relationship and value-dependent join)

(1) "Join rivers and towns along relationship lies-at"

$$\{rname(la.RIVER), rgeo(la.RIVER), tname(la.TOWN), tgeo(la.TOWN), tpopulation(la.TOWN) \mid la : lies\text{-}at\}$$

(2) "Join countries and towns such that the town's population is greater than the country's population"

$$\{tname(t), tpopulation(t), cname(c), cpopulation(c) \mid t : TOWN \wedge c : COUNTRY \wedge tpopulation(t) > cpopulation(c)\}$$

This completes our job: the foundation, walls, and roof of the EER house are finished; the reader is invited to move into it.

5. CONCLUDING REMARKS

After looking at the results achieved in this paper, we feel that the notion of an (extended) entity-relationship schema is now much better understood. Basically, we structured the schema into a data layer and an entity layer. We allowed for arbitrary user-defined data types, including the possibility of building new types and the use of aggregate functions such as SUM, MAX, or MIN. A precise mathematical semantics for this data layer has been presented. We extended the "classical" notion of an entity-relationship schema by allowing entities to have entities as components and to be constructed from entities. A (database) state for such a schema associated precise mathematical items with all parts of the schema, i.e., for the entity types, relationships, attributes, components, and constructions, permitting (especially) attributes and components to be optional.

Our (extended) entity-relationship calculus introduced and precisely defined the notions of term, range, declaration, and formula. In particular, we have allowed the declaration of new variables by means of arbitrary set-valued terms, and therefore had to give up the usual hierarchical structure of (relational) calculus. Queries were then seen as certain kinds of data type terms, together with a qualifying formula. By means of aggregate functions of the data layer, we were able to formulate queries not expressible, for instance, in the usual relational algebra or relational calculus. Nevertheless, our calculus preserved nice properties of the relational calculus such as relational completeness and safeness. Correct results of queries involving aggregate functions were made possible by strictly distinguishing between sets and bags (i.e., multisets).

Future work will use our calculus to define the semantics of query languages not only on the basis of the entity-relationship model, but also consider languages for the functional or the relational model. We shall apply the calculus to formulate integrity constraints restricting the possible database states. We are especially going to study the following properties of the EER model [32]:

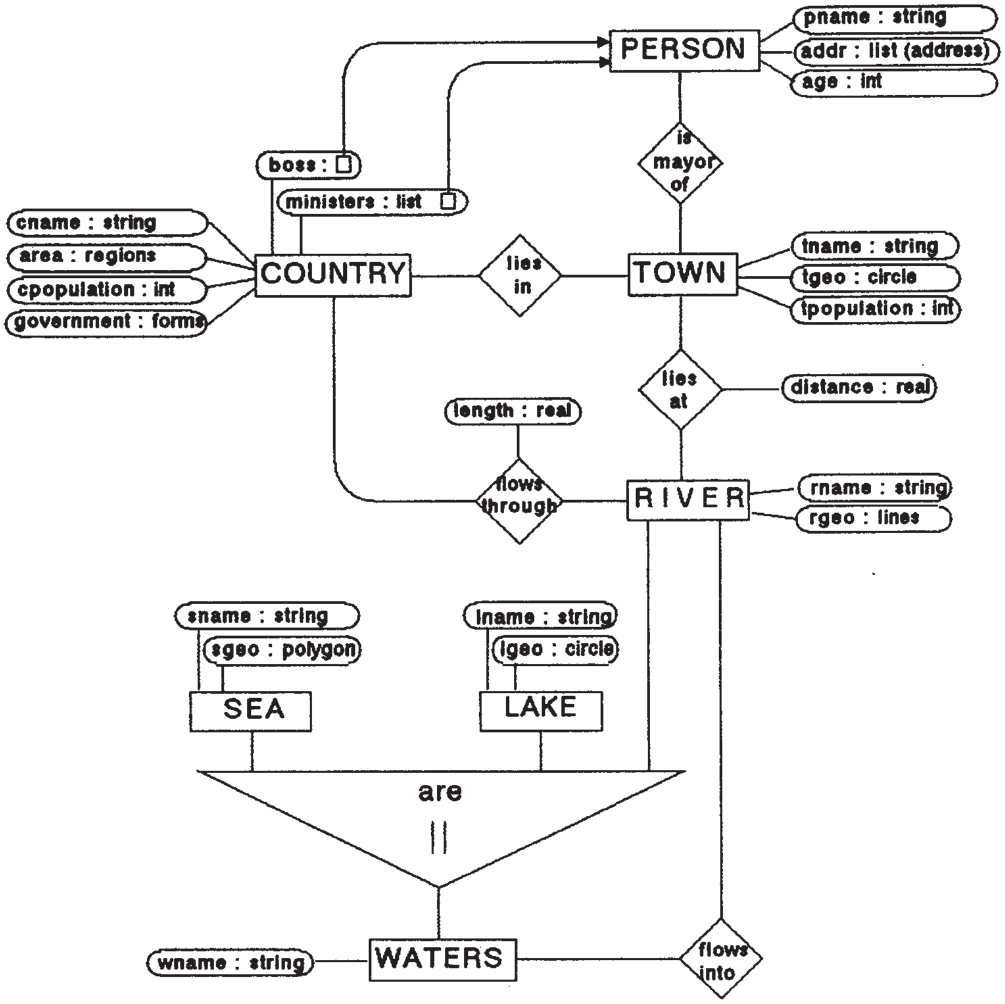
—key specifications for entity types,

—functional relationships,

—cardinality constraints for relationships, and set-, list-, and bag-valued attributes and components,
—nonoptional attributes, components, relationship participation, and
—derived attributes and components.

In our approach these properties can be derived as syntactic sugar [28a].

APPENDIX



ACKNOWLEDGMENTS

The critical comments of the referees have encouraged and helped us to improve the quality—in particular the readability—of this paper. Special thanks go to Udo (Walter) Lipeck for fruitful discussions and constructive criticism over a long period of time. Thanks also to our colleagues, especially Perdita Löhr-Richter, Karl Neumann, and Gunter Saake for commenting on draft versions of this paper. Special thanks to *07.12.56 for improving the English. Our ideas were born in the stimulating atmosphere of Hans-Dieter Ehrich's research group.

REFERENCES

1. ATZENI, P., BATINI, C., LENZERINI, M., VILLANELLI, F. INCOD: A system for conceptual design of data and transactions in the entity-relationship model. In *Proceedings of the 2nd International Conference on Entiy-Relationship Approach to Information Modelling and Analysis* (Los Angeles, Calif., Oct. 1981), pp. 375-410.
2. ATZENI, P., AND CHEN, P. P. Completeness of query languages for the entity-relationship model. In *Proceedings of the 2nd International Conference on Entity-Relationship Approach to Information Modelling and Analysis* (Los Angeles, Calif., Oct. 1981), pp. 109-122.
3. ABITEBOUL, S., AND HULL, R. IFO—A formal semantic database model. *ACM Trans. Database Syst. 12* (4) (1987), 525-565.
4. BIRD, R. S. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design.* Nato ASI Series, vol. F36, M. Broy, Ed. Springer-Verlag, Berlin, 1987, pp. 5-42.
5. *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages.* M. L. Brodie, J. Mylopoulos, J. W. Schmidt, Eds. Springer-Verlag, Berlin, 1984.
6. BRODIE, M. L., AND RIDJANOVIC, D. On the design and specification of database transactions. In *On Conceptual Modelling – Perspectives from Artificial Intelligence, Databases, and Programming Languages.* Springer-Verlag, Berlin, 1984, pp. 277-306.
7. BÜLTZINGSLOEWEN, G. v. Translating and optimizing SQL queries having aggregates. In *Proceedings of the 13th International Conference on VLDB*, P. M. Stocker, W. Kent, Eds. Brighton, 1987, pp. 235-243.
8. *Methodology and Tools for Database Design.* S. Ceri, Ed. North-Holland, Amsterdam, 1983.
9. CAMPBELL, D. M., EMBLEY, D. W., AND CZEJDO, B. A relationally complete query language for an entity-relationship model. In *Proceedings of the 4th International Conference on Entity-Relationship Approach* (Chicago, Ill., Oct. 1985), pp. 90-97.
10. CHEN, P. P. The entity-relationship model—Towards a unified view of data. *ACM Trans. Database Syst. 1*, 1 (Mar. 1976), 9-36.
11. CHEN, P. P. An algebra for a directional binary enity-relationship model. In *Proceedings of the International Conference on Data Engineering* (Los Angeles, Calif. April 1984), pp. 37-40.
12. DEMO, B., DILEVA, A., AND GIOLITO, P. An entity-relationship query language. In *Proceedings IFIP Workshop Conference on TFAIS* (Sitges, Spain, April 1985), A. Sernadas, J. Bubenlko, and A. Olive, Eds. North-Holland, Amsterdam, 1985, pp. 19-32.
13. DOS SANTOS, C. S., NEUHOLD, E. J., AND FURTADO, A. L. A data type approach to the entity-relationship approach. In *Proceedings of the 1st International Conference on Entity-Relationship Approach to Software Engineering* (Los Angeles, Calif., Dec. 1979). P. P. Chen, Ed. North-Holland, Amsterdam, 1980, pp. 103-119.
14. EHRICH, H.-D., DROSTEN, K., AND GOGOLLA, M. Towards an algebraic semantics for database specification. In *Proceedings IFIP TC2 Working Conference on Knowledge & Data "DS-2"* (Albufeira, Portugal, Nov. 1986), pp. 119-135.
15. EHRICH, H.-D. Key extensions of abstract data types, final algebras, and database semantics. In *Proceedings Workshop on Category Theory and Computer Programming* (LNCS 240, Springer-Verlag, Berlin, 1986), D. Pitt, S. Abramsky, A. Poigue, D. Rydehead, Eds. pp. 412-433.
16. EDER, J., KAPPEL, G., TJOA, A. M., AND WAGNER, R. R. BIER: The behavior integrated entity relationship approach. In *Proceedings of the 5th International Conference on Entity-Relationship Approach* (Dijon, France, Nov. 1986), pp. 147-166.
17. ELMASRI, R. A., AND LARSEN, J. A. A graphical query facility for ER databases. In *Proceedings of the 4th International Conference on Entity-Relationship Approach* (Chicago, Ill., Oct. 1985), pp. 236-245.
18. EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification I.* Springer-Verlag, Berlin, 1985.
19. IN *Proceedings of the 1st International Conference on Entity-Relationship Approach to Systems Analysis and Design* (Los Angeles, Calif., Dec. 1979), P. P. Chen, Ed. North-Holland, Amsterdam, 1980.
20. IN *Proceedings of the 2nd International Conference on Entity-Relationship Approach to Information Modelling and Analysis* (Los Angeles Calif., Oct. 1981), P. P. Chen, Ed. North-Holland, Amsterdam, 1983.

21. IN *Proceedings of the 3rd International Conference On Entity-Relationship Approach to Software Engineering* (Anaheim, Calif., Oct. 1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, Eds. North-Holland, Amsterdam, 1983.
22. IN *Proceedings of the 4th International Conference On Entity-Relationship Approach* (Chicago Ill., Oct. 1985).
23. IN *Proceedings of the 5th International Conference On Entity-Relationship Approach* (Dijon, France, Nov. 1986).
24. IN *Proceedings of the 6th International Conference On Entity-Relationship Approach* (New York, 1987), S. March, Ed.
25. IN *Proceedings of the 7th International Conference On Entity-Relationship Approach* (Rome, 1988).
26. IN *Proceedings of the 8th International Conference On Entity-Relationship Approach* (Toronto, Canada, 1989).
27. ELMASRI, R., AND WIEDERHOLD, G. GORDAS: A formal high-level query language for the entity-relationship model. In *Proceedings of the 2nd International Conference On Entity-Relationship Approach to Information Modelling and Analysis* (Los Angeles, Calif., Oct. 1981), P. P. Chen, Ed. North-Holland, Amsterdam, 1983, pp. 49-72.
28. ELMASRI, R., WEELDREYER, J., AND HEVNER, A. The category concept: An extension to the enity-relationship model. *Data Knowledge Eng. 1* (1985), 75-116.
28a. GOGOLLAS, M. Algebraization and integrity constraints for an extended entity-relationship approach. In *Proceedings TAPSOFT '89* (Barcelona, Spain 1989), Springer, LNCS 3S1. J. Diaz, Ed., pp. 259-274.
29. HENDRIX, G. G. Expanding the utility of semantic networks through partitioning. In *Proceedings Fourth International Joint Conference On Artificial Intelligence*, 1975, pp. 42-49.
30. HULL, R., AND KING, R. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv. 19*, 3 (1987), 201-260.
31. HAMMER, M., AND MCLEOD, D. Database description with SDM: A semantic data model. *ACM Trans. Database Syst. 6,* 3 (1981) 351-386.
31a. HOBENSTEIN, U. Automatic transformation of an entity-relationship query language into SQL. In *Proceedings of the 8th International Conference on Entity-Relationship Approach* (Toronto, Canada 1989), F. Lodeovsky, Ed., pp. 129-148.
32. HOHENSTEIN, U., NEUGEBAUER, L., SAAKE, G., AND EHRICH, H. -D. Three-level specification using an extended entity-relationship Model. In *Informationsbedarfsermittlung und -analyse für den Entwurf von Informationssystemen* (Informatik-Fachberichte Band 143, Springer, 1987), R. R. Wagner, R. Traunmüller, H. C. Mayr, Eds. pp. 58-88.
33. JAESCHKE, G., AND SCHEK, H.-J. Remarks on the algebra of nonfirst normal form relations. In *Proceedings of the 1st ACM SIGACT-SIGMOD Symposium On Principles of Database Systems* (Los Angeles, Calif., Mar. 1982), pp. 124-146.
34. KLUG, A. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM. 29*, 3 (July 1982), 699-717.
35. LIPECK, U. W., AND NEUMANN, K. Modelling and manipulating objects in geoscientific databases. In *Proceedings of the 5th International Conference On Entity-Relationship Approach* (Dijon, France, Nov. 1986), pp. 67-86.
36. LENZERINI, M., AND SANTUCCI, G. Cardinality constraints the entity-relationship model. In *Proceedings of the 3rd International Conference On Entity-Relationship Approach to Software Engineering* (Anaheim, Calif., Oct. 1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, Eds. North-Holland, Amsterdam, 1983, pp. 529-549.
37. MAIER, D. *The Theory of Databases*. Computer Science Press, Rockville, MD, 1983.
38. MARKOWITZ, V. M. *ERROL revisited: Toward a natural language oriented data description and manipulation language*. Tech. Rep. 413, Israel Institute of Technology, Haifa, 1986.
39. MARKOWITZ, V. M., AND RAZ, Y. A modified algebra and its use in an entity-relationship environment. In *Proceedings of the 3rd International Conference On Entity-Relationship Approach to Software Engineering* (Anaheim, Calif., Oct. 1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, Eds. North-Holland, Amsterdam, 1983, pp. 315-328.
40. MARKOWITZ, V. M., AND RAZ, Y. ERROL: An entity-relationship, role oriented query language. In *Proceedings of the 3rd International Conference On Entity-Relationship Approach to Software Engineering* (Anaheim, Calif., Oct. 1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, Eds. North-Holland, Amsterdam, 1983, pp. 329-345.

41. Myopoulos, J., and Wong, H. K. T. Some features of the TAXIS data model. In *Proceedings of the 6th International Conference On VLDB* (Montreal, Canada, Oct. 1980), pp. 399-410.
42. Özsoyoglu, G., Özsuyoglu, Z. M., and Matos, V. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987) 566-592.
43. Pirotte, A. High level data base query languages. In *Logic and Databases*. H. Gallaire, J. Minker, Eds. Plenum Press, New York, 1979, pp. 409-436.
44. Poonen, G. CLEAR: A conceptual language for entities and relationships. In *Centralized and Distributed Systems*. IEEE Computer Society. W. Chu, P. P. Chen, Eds. Silver Springs, Md., 1980, pp. 194-215.
45. Parent, C., and Spaccapietra, S. An entity-relationship algebra. In *Proceedings of the International Conference On Data Engineering*. (Los Angeles, Calif., April 1984). pp. 500-507.
46. Parent, C., and Spaccapietra, S. An algebra for a general entity-relationship model. *IEEE Trans. Softw. Eng. 11*, 7 (July 1985), 634-643.
47. Roesner, W. DESPATH: An ER manipulation language. In *Proceedings of the 4th International Conference On Entity-Relationship Approach* (Chicago, Ill., Oct. 1985), pp. 72-81.
48. Ramm, I., Neumann, K., Lipeck, U. W., and Ehrich. H.-D. Eine Benutzerschnittstelle für geowissenschaftliche Datenbanken. Informatik-Bericht Nr. 85-08, TU Braunschweig, 1985.
49. Shoshani, A. CABLE: A language based on the entity-relationship model. Rep. UCID-8005, Computer Science and Applied Mathematics Dept., Lawrence Berkeley Laboratory, Berkeley, Calif., 1979.
50. Shipman, D. W. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (1981), 140-173.
51. Subieta, K., and Missala, M. Semantics of query languages for the entity-relationship model. In *Proceedings of the 5th International Conference On Entity-Relationship Approach* (Dijon, France, Nov. 1986), S. Spaccapietra, Ed., pp. 197-216.
52. Subieta, K. Denotational semantics of query languages. *Inf. Syst. 12*, 3 (1987), 69-82.
53. Smith, J. M., and Smith, D. C. P. Database abstractions: Aggregation and generalization. *ACM Trans. Database Syst. 2*, 2 (1977), 105-133.
54. Schek, H.-J., and Scholl, M. H. The relational model with relation-valued attributes. *Inf. Syst. 11*, 2 (1986), 137-147.
55. Sernadas, A., Sernadas, C., and Ehrich, H.-D. Object-oriented specification of databases: An algebraic Approach. In *Proceedings of the 13th International Conference on VLDB* (Brighton, 1987), P. M. Stocker, W. Kent, Eds. pp. 107-116.
56. Schrefl, M., Tjoa, A. M., and Wagner, R. R. Comparison criteria for semantic data models. In *Proceedings International Conference On Data Engineering* (Los Angeles, Calif., 1984), pp. 120-125.
57. Teorey, F. J., and Fry, J. P. *Design of Database Structures*. Prentice-Hall, Englewood Cliffs, N. J., 1982.
58. Turner, R., and Lowden, B. G. T. An introduction to the formal specification of relational query languages. *Comput. J. 28*, 2 (1985), 162-169.
59. Teorey, T. J., Yang, D., and Fry, J. P. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Comput. Surv. 18*, 2 (1986), 197-222.
60. Ursprung, P., and Zehnder, C. A. HIQUEL: An interactive query language to define and use hierarchies. In *Proceedings of the 3rd International Conference On Entity-Relationship Approach to Software Engineering* (Anaheim, Calif., Oct. 1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh, Eds. North-Holland, Amsterdam, 1983, pp. 299-314.
61. Velez, F. LAMBDA: An entity-relationship based query language for the retrieval of structured documents. In *Proceedings of the 4th International Conference On Entity-Relationship Approach* (Chicago, Ill., Oct. 1985), pp. 82-89.
62. Zhang, Z. Q., and Mendelzon, A. O. A graphical query language for entity-relationship databases. In *Proceedings of the 3rd International Conference On Entity-Relationship Approach to Software Engineering* (Anaheim, Calif., Oct. 1983), C. G. Davis, S. Jajodia, P. A. Ng, and R. T. Yeh Eds. North-Holland, Amsterdam, 1983, pp. 441-448.

Received June 1988; final revision May 1990; accepted June 1990