Explicit Graphs in a Functional Model for Spatial Databases

Martin Erwig
erwig@fernuni-hagen.de

Ralf Hartmut Güting
gueting@fernuni-hagen.de

FernUniversität Hagen, Praktische Informatik IV,
58084 Hagen, Germany

Abstract. Observing that networks are ubiquitous in applications for spatial databases, we define a new data model and query language that especially supports graph structures. This model integrates concepts of functional data modeling with order-sorted algebra. Besides object and data type hierarchies graphs are available as an explicit modeling tool, and graph operations are part of the query language. Graphs have three classes of components, namely nodes, edges, and explicit paths. These are at the same time object types within the object type hierarchy and can be used like any other type. Explicit paths are useful because "real world" objects often correspond to paths in a network. Furthermore, a dynamic generalization concept is introduced to handle heterogeneous collections of objects in a query. In connection with spatial data types this leads to powerful modeling and querying capabilities for spatial databases, in particular for spatially embedded networks such as highways, rivers, public transport, and so forth. We use multi-level order-sorted algebra as a formal framework for the specification of our model. Roughly spoken, the first level algebra defines types and operations of the query language whereas the second level algebra defines kinds (collections of types) and type constructors as functions between kinds and so provides the types that can be used at the first level.

1 Introduction

We describe a new data model that can be seen from the following two perspectives:

(1) From an application point of view, it is a *model for spatial databases*. Its novel aspects are support for the modeling and querying of *networks* (highways, rivers, power lines, and so forth) and for queries on *heterogeneous collections of spatial objects*.

(2) From a general data modeling point of view, we provide *explicit graphs* as a modeling concept and integrate them into data model and query language. Also, heterogeneous collections of objects are cleanly modeled by a *dynamic generalization* concept.

The main emphasis lies on the first point. However, since we provide these features by introducing new concepts we also have to discuss data model aspects on a more general level. Let us briefly consider each perspective in turn.

Graphs and Heterogeneous Collections

We address two issues that have not been solved satisfactorily. The first is the modeling and querying of networks. Networks are a ubiquitous part of geographic information, for example,

- highways, roads, pedestrian ways, bicycle routes, trails, …
- rivers, lakes, canals, …
- trains, buses, underground, air planes, …
- electricity, telephone, gas, water, sewage, …

Current spatial database systems support well enough the handling of the geometry of such networks. That is, we may ask queries with respect to the spatial position of objects in a network, such as “Find roads intersecting a given region”. What is missing, is an adequate treatment of the connectivity, or graph structure, of the network. This is needed to support queries such as “Which parts of a river network are downstream from a pollution site and would be affected?” Other examples, that combine geometric and connectivity aspects, are “A fog region blocks part of the highway network. Which alternative routes should be recommended?” and “How many people live in the area affected by the failure of a power plant?”

The second issue concerns queries on distinct classes of objects that are in the query viewed under some common geometric aspect, like “Show everything within 50 kms from Munich” or “Which roads and supply lines (electricity, phone, …) intersect a planned highway section?” Current spatial data models support such queries – if at all – only by, for example, overlaying different query results on display devices. Generally, there are no clean modeling concepts for spatial data models to deal with heterogeneous collections of objects.

Data Modeling Aspects

Data models have to provide facilities to represent relationships among objects. In many cases it is

helpful to view such relationships as graphs structures: Then many queries can directly be mapped to well-known graph problems for which efficient algorithms exist (for example, route finding is solved by shortest paths). One approach is to consider certain structures of the data model as a graph (in the relational model, for example, a relation with two attributes, say, "from" and "to") and define graph operations which are only applicable to such structures [2, 6, 43]. We believe that this does not meet the importance of graphs in applications as sketched above. On the other hand, there are proposals to use explicit graph structures as the only modeling concept [10, 21]. However, this forces the user to re-model all other ("non-graph") relationships with graphs (this would be like forcing the user to encode spatial data types in relations).

In contrast, our approach is to offer graphs as a separate, "first-class" concept besides other facilities of a data model. Graphs are defined as explicit structures within a functional data model with an object class hierarchy. (Disregarding graphs, the functional model is similar to that of [46].) Spatial networks can be modeled in terms of graphs. Nodes and edges may carry geometric information, for example, a POINT value may be associated with a node and a polygonal LINE with an edge. Furthermore, explicit paths are available as entities in a graph. This is important since objects often correspond to paths in a network. For example, highways and rivers are naturally modeled as paths over the corresponding networks. Nodes, edges, and explicit paths are at the same time object classes of the hierarchy.

The implementation strategy behind this is to offer special data structures for the representation of graphs [25] that allow efficient traversal. Graph operations are to be implemented on the basis of efficient graph algorithms. The properties of spatially embedded networks can be used to obtain more efficient algorithms than is possible in the general case [36, 14].

Framework for Data Model Definitions

Since the model to be described has a sophisticated type system we find it convenient to use *multi-level order-sorted algebra* [16] as the formal basis to define all concepts within a common formalism.

Many-sorted algebra was introduced in [27] as a query language framework and further used in [22, 24, 5] to define a complex object algebra. The description of a query as a nested application of functions is common to both algebra and functional modeling. Many-sorted algebra provides in addition a well-structured type system. Order-sorted algebra [20] allows for subtype hierarchies and inheritance. Furthermore we can add a second level of order-sorted algebra to describe the applicability of type constructors to kinds, which are sets of sorts of the first level algebra, and a third algebra level formally accounts for operations with a variable number of arguments [16]. As a result one can model in a uniform framework a type system as well as a query algebra based on that type system. A similar framework with two levels is used in [26]. Order-sorted algebra allows one to represent data type as well as the object type hierarchies that are also part of functional modeling. Remaining key aspects of the functional model are the use of higher order functions and the modeling of object attributes and

relationships between objects by functions including multi-valued functions. It turns out that these concepts fit very well with the order-sorted algebra framework.

The paper is structured as follows: In Section 2 a short description of the framework of multi-level order-sorted algebra is given. The data model is then built along the concepts of data types, object types, and constructed types. Data and object type hierarchies and algebras are defined in Sections 3 and 4, respectively. In Section 5 constructed types and the central operations of the query language are described. The constructed types available are essentially sequences (Section 5.1) and graphs (Section 5.3). Operations to handle heterogeneous sequences of objects are defined in Section 5.2. After considering related work (Section 6) a brief discussion completes the paper (Section 7).

2 The Formal Framework: Multi-Level Algebra

We shall model a database system and its query language together with one specific database as a system of sets (representing data and object types) and functions (representing attributes and relationships) between these sets, that is, an algebra. This view is almost identical to that of the functional data model as described by [46].[1] We will use the facilities of multi-level order-sorted algebra to describe our data model and also specific schemas. In this section we briefly review the relevant notions and motivate the use of this formal system.

The classical relational algebra is a *universal* or *one-sorted* algebra: it has a single domain whose elements are relations and a collection of functions (such as join) on this domain. It has been recognized that it is advantageous for the definition of database query languages to move to a *many-sorted* algebra; for example, it is then possible to integrate arithmetic, aggregate functions and generally ADT functions into such an algebra [27, 22]. Furthermore, it would be nice to add the concepts of subtype and inheritance. From the algebra point of view, this means to define a (partial) subset order on algebra sets, which implies that functions defined on one set can be applied to elements of a subset. This is achieved by *order-sorted* algebra [20]. Complex object types are built from simpler ones by means of type constructors. The functions associated with generic type constructors are applicable to a large class of types, and their signatures typically contain type expressions in which constructors are applied to variables ranging over a set of type names. Such functions are said to be polymorphic. Since it is not possible, in general, to model parametric polymorphism by only one level of algebra [30] we can use a signature on a second level describing types and type constructors. The values of an algebra for such a signature are then considered as sort names on the first level. This idea, generalized to an arbitrary number of levels, is formalized by *multi-level* algebra [16]. A two-level algebra is used in [26] to describe a setting in which extensible database systems can be defined.

2.1 Modeling Subtypes and Inheritance with Order-Sorted Algebra

Let (S, Σ) be a partially ordered set. We assume that the order relation $\leq$ is extended in the canonical

[1] When we speak of *the* functional data model we mean just these basic features.

way to strings of equal length over S. The empty string is denoted by ε.

Definition 1. A *many-sorted signature* is a pair (S, Σ), where S is a set of *sorts* and Σ is an $S^* \times S$-sorted family $\{\Sigma_{w,s} \mid w \in S^* \text{ and } s \in S\}$ of *operations*. An operation $\sigma \in \Sigma_{w,s}$ is said to have *arity* w and *rank* ws. An *order-sorted signature* is a triple $(S, \leq, \Sigma)$ such that (S, Σ) is a many-sorted signature, $(S, \leq)$ is a partially ordered set, and the operations of Σ satisfy a monotonicity condition:

$$(\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'} \wedge w \leq w' \wedge w' \neq \varepsilon) \Rightarrow s \leq s'$$

An $(S, \leq, \Sigma)$-*algebra* A contains for each sort $s \in S$ a set s^A (the *carrier* of s) and for each operation symbol $\sigma_{w,s}$ a function $\sigma^A_{w,s}: w^A \to s^A$ where $(ws)^A = w^A \times s^A$ and $\varepsilon^A = \{\varnothing\}$. Algebra A must satisfy the following conditions:

(i) $s \leq s' \Rightarrow s^A \subseteq s'^A$

(ii) $\sigma \in \Sigma_{w,s} \cap \Sigma_{w',s'} \wedge a \in w^A \cap w'^A \Rightarrow \sigma^A_{w,s}(a) = \sigma^A_{w',s'}(a)$ □

Example. A many-sorted signature might have a sort set S = {REL, INT, LINE, …} and include operations

count:	REL	→	INT
intersects:	LINE × LINE	→	BOOL

An algebra for this signature would have a carrier set REL^A containing all possible relation instances, a carrier INT^A = {… -2, -1, 0, 1, 2, …}, and a function $\mathbf{count}^A$ mapping a relation to the number of tuples in it. In the sequel we shall use a function ‹ · › to refer to the meaning of a sort or operation, for example, INT^A will be denoted by ‹INT›. □

Data type and object type hierarchies can be defined by a partial order on sorts, see Figure 1.

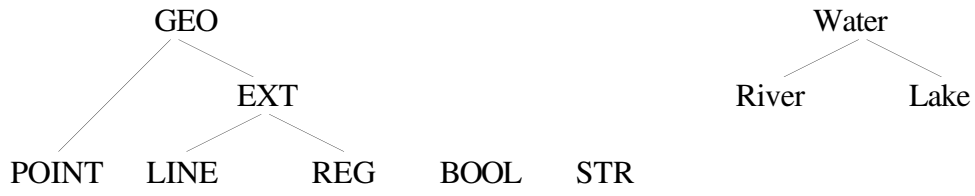

Figure 1

Operations are, for example,

inside:	POINT × REG	→	BOOL	(1)
inside:	GEO × REG	→	BOOL	(2)
name:	Water	→	STR	(3)
name:	River	→	STR	(4)
Rhine:		→	River	

The conditions above state, for example, that since POINT ≤ GEO we have ‹POINT› ⊆ ‹GEO› and

that the two **inside** functions agree on the ‹POINT›-subset of ‹GEO› (similarly for the function "name" on Water and River). Note that the definition of order-sorted algebra does not by itself specify that functions are inherited. Therefore we introduce signature specifications:

Definition 2. Any order-sorted signature $(S, \leq, \Sigma)$ is at the same time a *signature specification*. The *induced signature* $(S, \leq, IND(\Sigma))$ is defined by

$$IND(\Sigma) = \Sigma \cup \{\sigma_{w'', s} \mid \exists \sigma \in \Sigma_{w, s}: (w'' \leq w \wedge \forall \sigma \in \Sigma_{w', s'}: w' \leq w \Rightarrow w' < w'')\} \qquad \square$$

This definition describes inheritance of functions downwards: It says that we add σ on subtypes w'' of w with the same result type s as long as there is not already an overloaded definition of σ on w''.

From now on we shall interpret each signature as a signature specification. With regard to the above example this means that line (2) specifies five different functionalities (or, ranks) (including the one in line (1)) for **inside**. Likewise, we could have omitted line (4) since this signature entry is covered by line (3).

Note that terms, equations, congruence, and so on, are defined in the usual way. Equations will be used later to define abstract properties (such as associativity) of type constructors.

2.2 Decribing Parametric Polymorphism by a Second Algebra Level

A function having different types of a similar structure is said to be *parametric polymorphic*. This name reflects the fact that the ranks for this function are specified by one entry containing variables describing type parameters. In general, type variables range over a subset of all type names (for example, weak and equality variables in ML). Therefore, it is helpful to identify useful sets of type names and assign names to them. This can be done by defining a set of *kinds* [8].

Kinds are in fact sorts of another signature (on a second level), and their carriers are *sets of sorts of the first level*. Operations of the second level are called (type) *constructors*; the associated functions are mappings between kinds, that is, they map one or more sorts of one kind to a sort of another kind. Therefore we call this second level the *kind signature/algebra*. In addition, we need to specify the behaviour of constructors on the carriers of sorts. To see this, consider, for example, a cartesian product constructor which maps two sorts s and t to their product sort, prod(s, t). A reasonable property of the function prod^B (let B denote the corresponding kind algebra) would be, for instance, associativity, that is, the sorts prod(s, prod(t, u)) and prod(prod(s, t), u) are considered to be equal. This can be conveniently stated as a law/equation of the second level. In contrast, the intended mapping of the carriers is yet to be described by another function, say, $\overline{\text{prod}}$, and could be defined by: $\overline{\text{prod}}(s^A, t^A) = s^A \times t^A$. (Note that the carrier mapping must be compatible with the properties of prod^B). In the following definition we denote the individual algebra levels of a multi-level algebra by counting backwards (with regard to the construction history). That is, an $n{+}1^{\text{st}}$-level algebra A (or, A_1) depending on the n^{th}-level algebra B (or, A_2) is said to be on the first level whereas B is said to be on second level, and so on.

Definition 3. An order-sorted signature is a 1^{st}-*level signature*, and an order-sorted algebra is a 1^{st}-

level algebra. Given an n^{th}-level signature $(S, \leq, \Sigma)$ and an n^{th}-level $(S, \leq, \Sigma)$-algebra B, an order sorted signature $(S', \leq', \Sigma')$ is an $n+1^{st}$-*level signature* depending on $(S, \leq, \Sigma)$ and B if $S' = \bigcup_{s \in S} s^B$. An $(S', \leq', \Sigma')$-algebra A is an $n+1^{st}$-*level algebra* if for each operation $\sigma_{w, s} \in \Sigma$ there is a function $\overline{\sigma_{w, s}}$ (*type constructor* of $\sigma_{w, s}$) and if for each $t \in S'$ such that $t = \sigma^B_{w, s}(t_1, \ldots, t_n)$ (with $w = s_1 \ldots s_n$ and $t_i \in s_i^B$ for $1 \leq i \leq n$) we have $t^A = \overline{\sigma_{w, s}}(t_1^A, \ldots, t_n^A)$. The functions $\overline{\sigma_{w, s}}$ define the *constructor semantics* for Σ, and we say that A *depends* on B and the constructor semantics for Σ. □

We extend the previous example by adding the data sorts INT and REAL and introduce at the second level the set of kinds S = {**NUM**, **ORD**, **GEO**, **DATA**, **OBJ**, **ANY**, **SEQ**} and a sequence constructor "seq" which is used to build sorts of kind **SEQ**. The elements of the remaining kinds are given by (sort) constants. This is summarized in the following second-level signature.[2]

ord **NUM** ≤ **ORD**; **ORD**, **GEO** ≤ **DATA**; **DATA**, **OBJ** ≤ **ANY**

tc	INT, REAL:		→ **NUM**
	BOOL, STR:		→ **ORD**
	POINT, LINE, REG, EXT, GEO:		→ **GEO**
	Water, River, Lake:		→ **OBJ**
	seq:	**ANY**	→ **SEQ**

The subtype hierarchy is also shown in Figure 2.

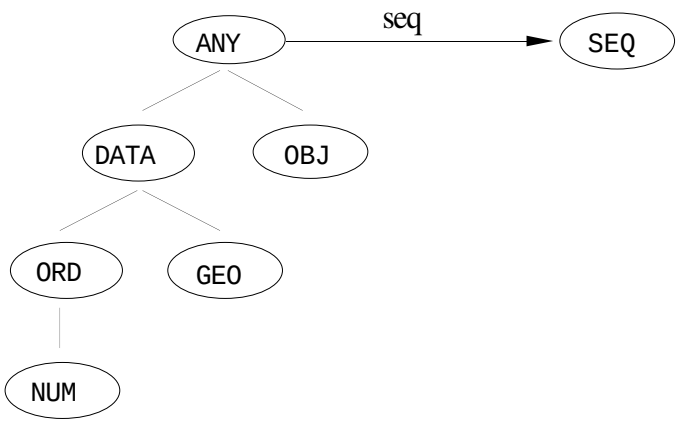

Figure 2

Since the term algebra is initial in the category of algebras we can use terms as representatives of values of an algebra for this signature, that is, we have (at the second level we use double angle brackets to denote interpretation in an algebra):

«**NUM**» = {INT, REAL}

[2] The key word **ord** precedes an order specification where $s, t, \ldots \leq v, w, \ldots$ is a shortcut for $s \leq v$; $t \leq v$; $s \leq w$; $t \leq w$, … **tc** introduces type constructors; on the first level, operations are introduced by **op**. Below we will also use equations which are preceded by **eq**.

«**ORD**» = {BOOL, STR, INT, REAL}

…

«**SEQ**» is the set of sorts that can be obtained by applying the seq constructor to sorts in «**ANY**»; the sorts in **SEQ** are denoted by the corresponding terms of this algebra, hence we have

«**SEQ**» = {seq(BOOL), seq(INT), …, seq(Water), seq(River), seq(Lake)}

The constructor semantics for the seq is defined as:

$\forall\ s \in$ «**ANY**»: $\quad \overline{\text{seq}}(\langle s \rangle) := \langle s \rangle^* \quad (= \langle \text{seq}(s) \rangle)$

Since the kind **SEQ** contains just the sorts obtained by applying seq to sorts in **ANY** we also use seq(**ANY**) to denote **SEQ**. Furthermore, kinds that are subsets of **SEQ** can be specified by applying seq to subsets of **ANY**, such as seq(**DATA**), seq(**OBJ**), and so forth.

The types of polymorphic functions are usually described by *type schemes*, for example, the type of the operation **count** could now be specified by:

$\forall\ \alpha \in$ «**ANY**». **count**: seq(α) $\rightarrow$ INT

As an abbreviation for this we replace occurences of the variable by the quantifying term, that is:

count: $\quad$ seq(**ANY**) $\quad \rightarrow$ INT

(Recall that the type is the same as **SEQ** $\rightarrow$ INT.) The idea behind is to expand kinds in such a type specification by all sorts contained in the kind. When a kind appears more than once in such a specification, each occurence is interpreted as a new variable quantification, for example,

/: $\quad$ **NUM** $\times$ **NUM** $\quad \rightarrow$ REAL

defines one division operator for any combination of INT and REAL operands. Here we can observe that since we are dealing with bounded type variables this notation is not only capable of describing parametric polymorphism but also *ad hoc polymorphism* (also often called *overloading*). We find it very convenient to use one notation for expressing these different aspects of overloading.

It is often desirable to specify that from a given kind always the same element is chosen in different positions of a signature specification; we denote this by indexing the kinds with the same index. For example, comparison operators can be applied to any two objects of the same sort within the kind **ORD** (which describes data types with a total order):

<, ≤, ≥, >: $\quad$ $\mathbf{ORD}_i \times \mathbf{ORD}_i \quad \rightarrow$ BOOL

With regard to variable quantification this means only the first occurrence of a specific indexed kind introduces a new variable, the remaining only refer to one. An operation that concatenates any two sequences of the same type could be defined as follows.

concat: $\quad$ $\mathbf{SEQ}_i \times \mathbf{SEQ}_i \quad \rightarrow \mathbf{SEQ}_i$

which could equivalently be denoted as

concat: $\text{seq}(\mathbf{ANY}_i) \times \text{seq}(\mathbf{ANY}_i) \rightarrow \text{seq}(\mathbf{ANY}_i)$

But one may also restrict to specific types of sequences, as in

sum, **min**: $\text{seq}(\mathbf{NUM}_i) \rightarrow \mathbf{NUM}_i$

The example given in this section is in fact part of the type system, or kind algebra, that we develop in this paper. To give an impression and survey of what is to come, Figure 3 shows the actual type system to be defined. The parts of this diagram will be explained in the following sections.

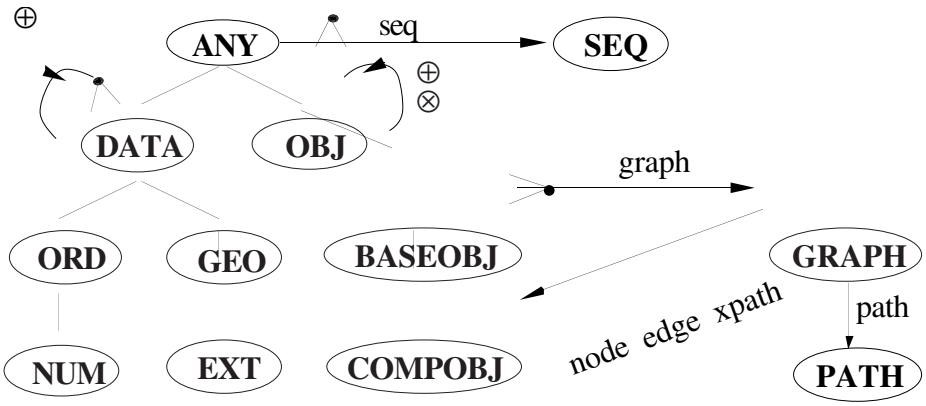

Figure 3

With multi-level algebra we have a fairly general tool to define languages with heavily overloaded operation symbols. So a few remarks on type checking are in order: Since in two-level signatures types are given by expressions containing bounded variables type checking can in principle be done by order-sorted unification [38] (see also [37, 29]). In our application we only have to check expressions of a query language, which means that signatures on both levels are fixed. Therefore we get along with a small subset of inference rules (in fact, we only need abstraction and application rules). Morevover, since we will require function definitions to be explicitly typed the type checker only needs pattern matching and not unification.

To close this section, let us summarize some points in favour of multi-level algebra. The use of multi-level algebra for the definition of our data model is motivated by:

- the convenient specification of all types of polymorphism (subtype, ad hoc, and parametric),
- a common framework for extending a data model with new structures (for instance, graphs, heterogeneous sequences),
- the ability to express subtle details for type constructors (for example, prevent nested applications of a sequence constructor),
- having available standard methods for static type checking.

3 Data Types

We shall develop the data model in three steps by introducing data types, then object types, and finally structures (or constructed types), which are sequences and graphs.

The data types used depend in general on the application to be supported. For our specific example application we introduce a collection of "standard" data types that are useful in many applications, and some geometric types. The sort hierarchy is shown in Figure 4 together with the part of the kind hierarchy (repeated from Figure 3) that is relevant here.

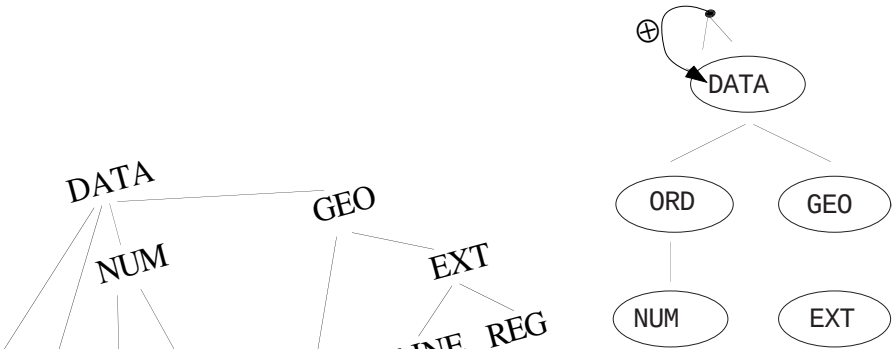

Figure 4

Let us first consider the sort hierarchy. The leaves of this tree show sorts for basic data types; we assume that for each of them the carrier is defined somewhere (for geometric types in [23]), but for this paper the precise definition is not relevant. An element of ‹LINE› is a polygonal line, an element of ‹REG› a simple polygon. The carrier of an internal node of the sort hierarchy is defined to be the union of the carriers of the children, for instance, ‹NUM› = ‹INT› ∪ ‹REAL›. This meets the subtype constraint of order-sorted algebra.

For the kind hierarchy, we assume that the carriers contain just the sorts that are descendants in the sort hierarchy, for example, «**NUM**» = {NUM, INT, REAL}, and for **ORD**, which is not a sort, that «**ORD**» = {BOOL, STR, INT, REAL}. Note, that the distinction between sort and kind hierarchy allows us to express rather subtle differences in the definition of operations. The definition

$$+: \quad \text{NUM} \times \text{NUM} \rightarrow \text{NUM}$$

specifies an addition operation that takes elements of the carrier of NUM and returns an element of this carrier; due to the sort hierarchy, this is also applicable to integers and reals. But the result type of this operation is NUM, even if it is applied to two integers. In contrast, the definition

$$+: \quad \mathbf{NUM}_i \times \mathbf{NUM}_i \rightarrow \mathbf{NUM}_i$$

specifies three addition operations with functionalities INT × INT → INT, REAL × REAL → REAL, and also the one above, NUM × NUM → NUM. Hence with this definition we know that "+" returns

an integer if it is applied to two integers. The following operations are defined for standard types.

op	**and**, **or**:	BOOL × BOOL	→	BOOL
	not:	BOOL	→	BOOL
	+, -, *, **div**, **mod**:	INT × INT	→	INT
	/:	INT × INT	→	REAL
	+, -, *, /:	REAL × **NUM**	→	REAL
	+, -, *, /:	**NUM** × REAL	→	REAL
	=, ≠:	$\mathbf{DATA}_i \times \mathbf{DATA}_i$	→	BOOL
	<, ≤, ≥, >:	$\mathbf{ORD}_i \times \mathbf{ORD}_i$	→	BOOL

For our example spatial application, we define the following operations (a more complete collection can be found in [22]):

op	**inside**:	**GEO** × REG	→	BOOL
	intersects:	**EXT** × **EXT**	→	BOOL
	intersection:	LINE × LINE	→	seq(POINT)
	intersection:	LINE × REG	→	seq(LINE)
	intersection:	REG × LINE	→	seq(LINE)
	intersection:	REG × REG	→	seq(REG)
	closest:	seq(POINT) × POINT	→	POINT
	concat:	LINE × LINE	→	LINE
	mindist:	**GEO** × **GEO**	→	REAL
	length:	LINE	→	REAL
	area:	REG	→	REAL

The **intersection** operator returns a set of intersection objects in the form of a sequence (the seq constructor is defined as in Section 2). Given a sequence of points *S* and another point *P*, the **closest** operator returns the point in *S* that is closest to *P*. (Usually, we will expect only one closest point; when there are more than one, one point is chosen non-deterministically.) The remaining operations are mostly self-explaning.

In the first place, the signature entries specify the typing of operations and not their syntax. In the following we will use the convention that binary predicates as well as arithmetic operations are written infix and that all other operations are written postfix. In addition, we allow any unary function to be used in prefix notation with its argument following in parentheses.

If we look carefully at the signature specification, it seems that the sort hierarchy is not really needed; the definitions just rely on the specification of kinds to achieve the desired form of overloading. Could we not omit the sort hierarchy?

Indeed, the sort hierarchy has another purpose, which is related to "dynamic generalization" discussed in the introduction and the ⊕ constructor in the kind algebra. If the sort hierarchy were omitted, then there would be a kind **GEO** but no sort GEO (there would be no carrier for GEO) and not a sort seq(GEO) either. However, we are interested in these generalization data types in connection with the handling of heterogeneous sequences and generalization of object types (see Section 5.2). For this purpose, the ⊕ constructor (union) on data types with functionality

$$\oplus: \quad \mathbf{DATA} \times \mathbf{DATA} \quad \rightarrow \quad \mathbf{DATA}$$

is defined to return for any two sorts in «**DATA**» their smallest common supersort, for example, LINE ⊕ REG = EXT and POINT ⊕ LINE = GEO. Introducing **DATA** not only as a kind but also as a sort makes ⊕ total on **DATA**.

4 Object Types

The part of the order-sorted algebra that we have seen so far, concerned with data types, models representation and querying facilities of a *database system*. The same holds for the structures, sequences, and graphs, that will be considered in Section 5. In contrast, the subalgebra dealing with object types models one specific *database*. Sorts represent object classes, and operations represent functions applicable to objects. As usual in functional modeling, these functions serve to describe object attributes as well as relationships between objects. Functions may be single- or multi-valued; if they are multi-valued, we represent the result set as a sequence of objects or data values.

Let us introduce an example database describing cities, highways, and so on. This database has a sort hierarchy, which is shown in Figure 5 together with the part of the kind algebra relevant to this section.

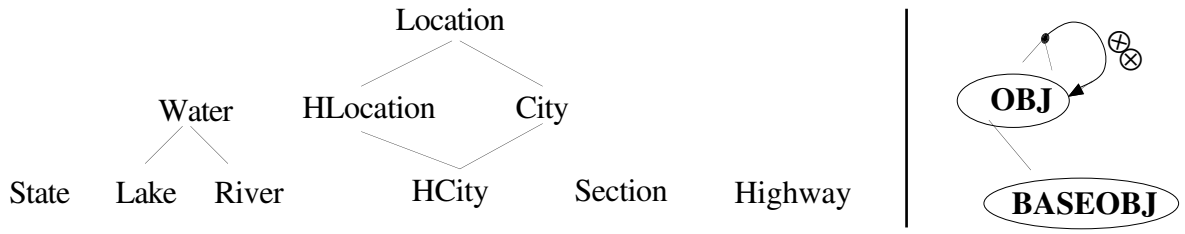

Figure 5

The object sorts shown model objects *stored in the database*; they make up the kind **BASEOBJ**:

«**BASEOBJ**» = {State, Water, Lake, River, Location, City, HLocation, HCity, Section, Highway}

There are also "potential objects" derivable from the database, which are not stored. These sorts are collected in a kind **OBJ** discussed below. For each sort in **BASEOBJ** the carrier is in principle a set of object identifiers, which are drawn from some unlimited domain Ω. For technical reasons that will

become clear shortly, instead of an object identifier we use a list containing just that single object identifier. So we have (let Ω^+ denote the set of non-empty sequences over Ω):

$$\forall\, s \in \text{«}\mathbf{BASEOBJ}\text{»}: \qquad \text{‹}s\text{›} \subseteq 2^{\Omega^+}$$

So each carrier is a set of lists of object identifiers. If there are three rivers stored in the database, the carrier of River contains three one-element lists:

$$\text{‹River›} = \{\langle r_1 \rangle, \langle r_2 \rangle, \langle r_3 \rangle\}$$

From the definition of order-sorted algebra we know that ‹River› $\subseteq$ ‹Water›. For object sorts we require additionally that each object has a single base type (together with its supertypes in the hierarchy) which can be formulated as follows:

$$\forall\, s, t \in \text{«}\mathbf{BASEOBJ}\text{»}:$$
$$\text{‹}s\text{›} \cap \text{‹}t\text{›} \neq \emptyset \;\Rightarrow\; \exists\, u \in \text{«}\mathbf{BASEOBJ}\text{»}: \quad u \leq s \;\text{ and }\; u \leq t \;\text{ and }\; \text{‹}s\text{›} \cap \text{‹}t\text{›} = \text{‹}u\text{›}$$

This says in particular, that an object cannot belong to two sorts that are different leaves of the sort hierarchy.

The operations defined on the object types of our example database are given in Figure 6. To enhance the readability of queries we write object sorts with a capital first letter.

name:	State	→	STR	name:	City	→	STR
region:	State	→	REG	pop:	City	→	INT
				facilities:	City	→	seq(STR)
name:	Water	→	STR	lies_in:	City	→	State
surface:	Lake	→	REG	way:	Section	→	LINE
				limit:	Section	→	INT
flow:	River	→	LINE	duration:	Section	→	REAL
pos:	Location	→	POINT	hno:	Highway	→	INT
				route:	Highway	→	LINE
				visits:	Highway	→	seq(State)

Figure 6

Recall that we regard this signature as a signature specification, thus we can assume that functions are inherited along the sort hierarchy which means that the function “name” is applicable to River and Lake objects and a City has a position available through the “pos” function, which it inherits from the Location sort that just has a POINT attribute. Note that there are object-valued and multi-valued functions. The object types HLocation, HCity, Section, and Highway are used below to model a highway

network. HLocation and HCity are specializations of Location and City, respectively, containing those positions and cities that are also nodes of the highway network; the HLocation type represents, for example, junctions or exits. Obviously, a HCity is a city but also a special type of location within the highway network. HLocations are connected by highway Sections; the geometry of the Section is given by the "way" attribute. A Highway corresponds to a path over this network. The relationships between HLocations, Sections, and Highways in the network are defined in Section 5.3 by a graph structure.

HCity being a subtype of two types, embodies what is often called *multiple inheritance*. Since the definition of order-sorted algebra requires that for an operation defined on more than one type, all associated functions of an algebra must agree on common subsets (Definition 1, (ii)) no ambiguities can arise. Of course, this condition must be ensured by an implementation.

The functions defined on object types may be *extensional* or *intensional;* extensional functions are stored in the database whereas intensional or *derived* functions are given by a defining expression. From the point of view of the algebra and for the user there is no difference between extensional and intensional functions except that extensional functions are restricted to have a single operand sort (this is mainly for implementation reasons). In fact, some of the functions listed above are defined intensionally. Derived functions can be built by means of lambda-abstraction. We do not allow recursive definitions, such as those that determine transitive relationships between objects. Such relationships should be modeled and queried using graphs, that is, we follow [10]: "recursive queries without recursion". Using a syntax similar to ML the definition of the derived function "duration" looks like:

fun duration(*x*:Section):REAL =
(*x* way **length**) / (*x* limit)

The type annotation is not really needed in this example since the function "way" is only defined on the type Section, and thus the type for "duration" could be inferred. To keep type checking simple (that is, to avoid ambiguity problems occurring in the type inference for overloaded functions) we require, though, each definition of a derived function to have a type annotation. The formal status of function objects in the algebra is explained in [17, 16].

Derived Objects

What remains to be explained about object types are the mysterious "potential objects" and the type constructors $\oplus$ and $\otimes$. The corresponding part of the second level signature is:

ord **BASEOBJ** $\leq$ **OBJ**
tc $\oplus$: **OBJ** $\times$ **OBJ** $\rightarrow$ **OBJ**
$\otimes$: **OBJ** $\times$ **OBJ** $\rightarrow$ **OBJ**
eq $s \oplus s = s$
$s \oplus t = t \oplus s$

$$s \oplus (t \oplus u) = (s \oplus t) \oplus u$$
$$s \otimes t = t \otimes s$$
$$s \otimes (t \otimes u) = (s \otimes t) \otimes u$$
$$s \otimes (t \oplus u) = (s \otimes t) \oplus (s \otimes u)$$

The equations specify that both type constructors are commutative and associative, that $\otimes$ distributes over $\oplus$, and that $\oplus$ is idempotent. In the following we explain these constructors in detail.

First, we would like to be able to form in a query the union of any two collections of objects of sorts s and t. We shall later introduce a corresponding union operation in the algebra. A sort resulting from this operation will represent a collection of objects of "union type" $s \oplus t$. Therefore, the kind **OBJ** contains a set of sorts that can be derived from the sorts in **BASEOBJ** by application of the $\oplus$ constructor. For our example database, we have

$$\{\text{State}, \text{State} \oplus \text{River}, \text{River} \oplus \text{City}, \text{River} \oplus \text{City} \oplus \text{Highway}\} \subseteq «\textbf{OBJ}»$$

Again, the sorts in **OBJ** are denoted by terms of the type algebra whereas the constructor semantics of $\oplus$ is

$$\forall\, s, t \in «\textbf{OBJ}»: \qquad ‹s› \mathbin{\overline{\oplus}} ‹t› \;:=\; ‹s› \cup ‹t› \qquad (= ‹s \oplus t›)$$

It follows from this definition that $\overline{\oplus}$ is idempotent, associative, and commutative, which meets the specification from above. In the object sort hierarchy, we shall view the constructed sort as a supersort of the operands to the constructor, for example, State $\leq$ State $\oplus$ River and River $\leq$ State $\oplus$ River:

$$\forall\, s, t \in «\textbf{OBJ}»: \qquad s \leq s \oplus t \wedge t \leq s \oplus t$$

The second constructor applicable to object types is the $\otimes$ (product) constructor. In a query it is sometimes necessary to form "aggregation objects", that is, to build from two objects o_1 and o_2 an aggregation object on which all functions applicable to o_1 *and* all functions applicable to o_2 are defined. In a relational setting, this corresponds to concatenating two tuples when forming a cartesian product; the result tuple has all attributes of both operand tuples. The $\otimes$ operation allows us to form a corresponding product object type (sort). Hence the carrier of State $\otimes$ City contains one object for each combination of objects in the carriers of State and of City. Formally, we define the following subsort relationships:

$$\forall\, s, t \in «\textbf{OBJ}»: \qquad s \otimes t \leq s \wedge s \otimes t \leq t$$

Now, the terms State $\otimes$ City, Highway $\otimes$ City $\otimes$ River, and so on, are also in **OBJ**. The constructor semantics is defined by aggregation objects which are built by concatenating the lists of object identifiers of the two operand objects (this is the reason why lists are used at all). For example, there may be an object $\langle h_{42}, c_{17} \rangle$ in the carrier of Highway $\otimes$ City and an object $\langle r_{92} \rangle$ in the carrier of River which would lead to an aggregation object $\langle h_{42}, c_{17}, r_{92} \rangle$. One can already observe that with this definition

$\overline{\otimes}$ is associative. In addition, we need it to be commutative. For this purpose, we define arbitrarily some total order on object sorts in **BASEOBJ**, for example,

State < Water < Lake < River < Location < City < HLocation < HCity < Section < Highway

and keep for all aggregation objects the list of object identifiers sorted according to this order. Hence the example objects would be $\langle c_{17}, h_{42} \rangle$ in the carrier of Highway ⊗ City and $\langle r_{92} \rangle$ in the carrier of River and the resulting aggregation object is $\langle r_{92}, c_{17}, h_{42} \rangle$. If one object sort occurs several times, as in $\text{River}_1 \otimes \text{City} \otimes \text{River}_2$ (like in a query "find the city closest to the point where two rivers meet") the order of writing is assumed to be preserved, so the object identifiers would be arranged according to the order $\text{River}_1 \otimes \text{River}_2 \otimes \text{City}$. Formally, let there be an operation *merge* that merges two ordered lists of object identifiers. Then

$$\forall\, s, t \in \text{«}\mathbf{OBJ}\text{»}: \quad \text{‹}s\text{›} \,\overline{\otimes}\, \text{‹}t\text{›} \;:=\; \{merge(s', t') \mid s' \in \text{‹}s\text{›}, t' \in \text{‹}t\text{›}\} \qquad (= \text{‹}s \otimes t\text{›})$$

Now, this definition is consistent with associativity and commutativity specified for ⊗ above. Furthermore, it is easy to check that $\overline{\otimes}$ distributes over $\overline{\oplus}$. Observe that we can form arbitrarily nested sort expressions with union and product type constructors. We have here the situation mentioned in Section 2 that there may be many sort expressions denoting the same carrier; due to the associativity, commutativity, and distributivity laws any such expression may be transformed into a "disjunctive normal form"

$$(s_{1,1} \otimes s_{1,2} \otimes \ldots \otimes s_{1,n_1}) \oplus \ldots \oplus (s_{k,1} \otimes s_{k,2} \otimes \ldots \otimes s_{k,n_k})$$

which allows to decide whether two expressions are equivalent.

5 Structured Types and the Query Language

The two fundamental structures available in this model are *sequences* and *graphs*. These are introduced through constructors "seq" and "graph", respectively. A database is modeled by an object type hierarchy together with a collection of graph definitions. Sequences are the central tool for data manipulation and a collection of sequence operations is offered as a "general purpose" part, called the *kernel operations*, of the query language algebra.

5.1 Sequences

In this subsection we describe the kernel operations of the query language (this collection might be extended if other operations are needed). The seq constructor was already defined in Section 2. Note that it can be applied only to sorts in **ANY**; it is not applicable to sorts in **SEQ** which means we have no nested sequences. A sequence of objects is made available to a query simply by writing down the name of an object type. For an object sort *s*, the result of writing "*s*" is a sequence containing all elements in ‹*s*› precisely once, in some unspecified order. As we have seen in Section 4, multi-valued

object functions and some data functions (**intersection**) also return sequences; so we can also obtain a sequence by applying such a function. We prefer to use sequences as a fundamental structure rather than sets because it is then possible to include sequence operations such as sorting, grouping, and so on into the algebra. This is motivated further in [27]. Next we give signature specifications for the central parts of the query language.

Select

The **select** operation takes one or more sequences of objects seq($\mathbf{OBJ}_1$), …, seq($\mathbf{OBJ}_k$) as operands. Conceptually, it then produces one aggregate object (see Section 4) of type $\mathbf{OBJ}_1 \otimes \ldots \otimes \mathbf{OBJ}_k$ for each combination of objects in the operand sequences. It further takes as a parameter a predicate applicable to objects of type $\mathbf{OBJ}_1 \otimes \ldots \otimes \mathbf{OBJ}_k$ and returns in the result sequence those aggregate objects for which the parameter predicate evaluates to *true*.

$$\textbf{select}: \text{seq}(\mathbf{OBJ}_1) \times \ldots \times \text{seq}(\mathbf{OBJ}_k) \times [\mathbf{OBJ}_1 \otimes \ldots \otimes \mathbf{OBJ}_k \rightarrow \text{BOOL}] \rightarrow \text{seq}(\mathbf{OBJ}_1 \otimes \ldots \otimes \mathbf{OBJ}_k)$$

With this functionality, **select** comprises the operations of selection, cartesian product, and join of the classical relational algebra; in fact, it is much more general, since it allows parameter predicates that include arbitrary expressions of the query algebra.[3] It can be used as comfortably as the select-from-where clause in SQL. This means that a user can easily translate his favourite SQL-queries into our language. Note, however, that projection is provided through a special **Show** command (see below) and that group-by must be reformulated using the inverse operation.

In the above signature entry we have used an obvious notation for denoting a sequence of sorts. This can be formalized by a third algebra level (called *class signature/algebra*), which is actually demonstrated in [16]. The formalization requires some technical effort, and for lack of space we describe only the idea how it can be performed: First, define a *class*, say, **TOP**, the carrier of which will contain all kinds that are allowed to appear in sequences. Second, define a *kind constructor* list: **TOP** $\rightarrow$ **LIST** with the constructor semantics being the same as for the "seq" constructor. Thus, for a kind k, «list(k)» contains all sequences of sorts from kind k. Using the notational conventions for type schemes introduced in Section 2, the expression list(**OBJ**) introduces the variable binding $\forall\, \alpha \in$ «list(**OBJ**)» and thus denotes sequences of object sorts, such as, ⟨Water, City, River⟩. Now, in order to denote a product type Water ⊗ City ⊗ River we need a means to "fold" the ⊗ constructor along the sequence, and the type seq(Water) ⊗ seq(City) ⊗ seq(River) is obtained by first "mapping" the "seq" constructor to the list and then folding ⊗. This means, having a function *map* which applies a type constructor to each element in a list of types and a function *fold* which reduces a list of types by a binary type constructor we can then specify the type of **select** by:

[3] These are elements of functional sorts which are explained in [17].

select: $fold(\times, map(\text{seq}, \text{list}(\mathbf{OBJ})_i)) \times [fold(\otimes, \text{list}(\mathbf{OBJ})_i) \rightarrow \text{BOOL}]$
$\rightarrow \text{seq}(fold(\otimes, \text{list}(\mathbf{OBJ})_i))$

For readability we shall, however, use the intuitive notation from above. Yet, another new notation occurs in the signature: Sometimes we write operand sorts in square brackets. This is to specify that in queries arguments are written in square brackets behind the operator. Arguments of these sorts are also called *parameters*. For example,

River **select**[(flow **length**) > 100]
Show name, flow

asks for all rivers longer than 100 kilometers. The predicate of **select** compares the length of a river which is given by a composition of the functions flow: River → LINE and **length**: LINE → REAL with a constant. Note that we have made a notational simplification: We are able to omit lambda-abstraction (see [17]) since we know that **select** takes a function defined on objects of type given by the operand sequence. This means that a parser can always add the missing parts; in this example the full expression would be:

River **select**[**fun** (*r*) ⇒ (*r* flow **length**) > 100]

To this query we have added a **Show** command in order to "see" some aspects of the returned River objects. We assume that the task of a query is to return a collection of objects. An object can be made visible only by applying some functions to it that return data values, for which a textual or graphical representation exists. This is performed by the **Show** command that takes a list of function names and applies these functions to all objects resulting from a query (therefore there is no "projection" operation in the algebra). Since **select** can be applied to an arbitrary number of object sequences we may ask for all bridges of highways across rivers by:

Highway River **select**[route **intersects** flow]

Since aggregate objects formed by **select** preserve all functions defined on their component objects we must be able to deal with name clashes between these functions. This is done by supplying additional names for the object sorts in a query which have to be appended to the conflicting functions:

Highway _H River _R **select**[route **intersects** flow]
Show name_H, name_R

We have stated above that **select** returns in the result sequence those aggregate objects for which the parameter predicate evaluates to *true*. It is convenient to allow the parameter predicate to be undefined for a given aggregate object (without raising an error condition) and to define the semantics of **select** in such a way that these objects are simply omitted from the result sequence. For example, suppose we obtain as the result of a subquery some set of Location objects "QueryLocation"; among these, we

are only interested in big cities (that is, cities with population larger than 500 000). Though "Location" does not have a "pop" attribute one can just write:

QueryLocation **select**[pop > 500000]

But note that the result of this query is still of type seq(Location).

Sequence Transformers

The following operations manipulate sequences:

op	**map**:	$\text{seq}(\mathbf{ANY}_i) \times [\mathbf{ANY}_i \to \mathbf{ANY}_j]$	$\to$	$\text{seq}(\mathbf{ANY}_j)$
	map:	$\text{seq}(\mathbf{ANY}_i) \times [\mathbf{ANY}_i \to \text{seq}(\mathbf{ANY}_j)]$	$\to$	$\text{seq}(\mathbf{ANY}_j)$
	asc, **desc**:	$\text{seq}(\mathbf{ANY}_i) \times [\mathbf{ANY}_i \to \mathbf{ORD}]$	$\to$	$\text{seq}(\mathbf{ANY}_i)$
	head, **tail**:	$\text{seq}(\mathbf{ANY}_i) \times [\text{INT}]$	$\to$	$\text{seq}(\mathbf{ANY}_i)$
	rdup:	$\text{seq}(\mathbf{ANY}_i)$	$\to$	$\text{seq}(\mathbf{ANY}_i)$

The **map** operation, with its first functionality, takes a sequence of elements of some sort *s* in **ANY** and a function from *s* to some *t* in **ANY** and returns a sequence of elements of *t*. It produces the result sequence by applying the parameter function to each element of the operand sequence. We might compute the total length of the rivers in River as follows:

River **map**[flow **length**] **sum**

The parameter function of **map** is here a composition of the functions flow: River → LINE and **length**: LINE → REAL. Hence the result of applying **map** is of type seq(REAL) to which the aggregate function **sum** (see below) can be applied, producing a result of type REAL.

Even with the type $\mathbf{ANY}_i \to \mathbf{ANY}_j$ of parameter functions for **map** it is possible to use sequence-valued functions if they are composed with other functions to yield the required function type. For example, we may determine the maximal number of states "visited" by a highway, as follows:

Highway **map**[visits **count**] **max**

On the other hand, it is also possible to use indeed a sequence-valued parameter function; this is the second functionality of **map** with parameter function type $\mathbf{ANY}_i \to \text{seq}(\mathbf{ANY}_j)$. In this case **map** concatenates all the result sequences obtained by applying the parameter function to elements of the operand sequence. We might use this to get a list of all states visited by any highway:

Highway **map**[visits] **rdup**

Here we have additionally used the **rdup** ("remove duplicates") operation to obtain each state only once. The operations **asc** and **desc** are for sorting sequences. They take a parameter function mapping the elements of the sequence into one of the totally ordered domains in **ORD** according to which the sequence is to be sorted. **Head** and **tail** return a front or end part of a sequence. For example, we might

get an alphabetic list of the ten largest states by the query:

State **desc**[region **area**] **head**[10] **asc**[name]
Show name

For simplicity we have defined **asc** and **desc** with just one parameter function, this could be extended to a list of functions to allow lexicographic sorting by several attributes.

Aggregate Functions

An aggregate function combines the elements of a sequence into an atomic value. We already used the functions **max** and **sum**, which are instances of the more general operation **agg**.

op	**agg**:	$\text{seq}(\mathbf{ANY}_i) \times [\mathbf{ANY}_i \times \mathbf{ANY}_i \to \mathbf{ANY}_i]$	$\to$	$\mathbf{ANY}_i$
	sum:	$\text{seq}(\mathbf{NUM}_i)$	$\to$	$\mathbf{NUM}_i$
	min, **max**:	$\text{seq}(\mathbf{ORD}_i)$	$\to$	$\mathbf{ORD}_i$
	count:	$\text{seq}(\mathbf{ANY})$	$\to$	INT
	exists, **forall**:	$\text{seq}(\mathbf{ANY}_i) \times [\mathbf{ANY}_i \to \text{BOOL}]$	$\to$	**BOOL**
	the:	$\text{seq}(\mathbf{ANY}_i)$	$\to$	$\mathbf{ANY}_i$
	in:	$\text{seq}(\mathbf{ANY}_i)$	$\to$	$(\mathbf{ANY}_i \to \text{BOOL})$

The **agg** function is a fairly general tool to build aggregate functions. It takes as a parameter a binary function applicable to elements of its operand sequence (which must not be empty) and extends this function to return a value for the whole sequence. For a sequence containing only one element just that element is returned.[4] We are interested in **agg** to produce non-standard aggregate functions as in the following example. A highway is a path over the highway network and as such consists of a sequence of edges (highway sections) each of which has an associated LINE attribute. With **agg** we can compute the LINE value of the complete highway which is done by the derived function "route":

fun route(*h*:Highway):LINE =
  h **path edges map**[way] **agg**[**concat**]

(The subexpression "*h* **path edges**" forms the sequence of edges of the path associated with highway object *h*, see Section 5.3.)

The aggregate function **the** is used to extract single objects from sequences. It is defined only for sequences containing one element and returns just this element. For example, the object representing the river "Rhine" can be retrieved by

Rhine = **the**(River **select**[name = 'Rhine'])

[4] Note that **agg** is a simplified version of the *fold* operation found in functional languages, such as, ML: First, the type of parameter functions is a special case of the more general type that is possible, namely $\alpha \times \beta \to \beta$. Second, we have omitted the default value for empty sequences since this makes queries cumbersome to read.

Note that this is defined only if there exists exactly one river with that name. The operator **the** makes it easy to define the derived function "lies_in":

fun lies_in(*c*:City):State =
    the(State **select**[(*c* location) **inside** region])

Another derived function is "visits" defined as follows:

fun visits(*h*:Highway):seq(State) =
    State **select**[(*h* route) **intersects** region]

The **in** operation makes it possible to form set differences: Given a sequence *S* of type seq(*t*), it returns a function of type (*t* → BOOL) that can be used to check whether any object of type *t* is contained in *S*. For example, suppose we would like to determine the states *not* visited by any highway. Applying **in** to the sequence "HighwayStates" yields a function of type (State → BOOL) that allows to check for each State whether it is an element of the collection HighwayStates. Such constructed functions can be used as parameter functions:

State **select**[**in**(HighwayStates) **not**]
City **select**[lies_in **in**(HighwayStates)]

Inverse Operation

The inverse operation is an important element of the functional data model; while a function realizes a relationship with a predefined direction of access, the inverse operation allows for querying the relationship in the opposite direction.

inv: $[\mathbf{OBJ}_i \rightarrow \mathbf{ANY}_j] \rightarrow (\mathbf{ANY}_j \rightarrow \text{seq}(\mathbf{OBJ}_i))$

Applied to a mapping *f*: *s* → *t* that associates with an object of type *s* an object or value of type *t*, **inv** returns a function of type *t* → seq(*s*) that yields for each object or value in *t* the list of elements of *s* that are mapped into it by *f*. Note that **inv** may even be applied to derived object functions: Since object types will be finite a single-valued derived function can always be materialized, and from that the inverse function can easily be constructed (of course, in some cases optimization will yield better results.) For example, we may define as a derived function the set of cities belonging to a state:

fun cities(*s*:State):seq(City) =
    s **inv**[lies_in]

This function can then be used in a query "List for each state the number of its big cities!":

fun no_big_cities(*s*:State):INT =
    s cities **select**[pop > 500000] **count**
State
Show name, no_big_cities

5.2 Heterogeneous Sequences and Dynamic Generalization

We shall now use the union type introduced in Sections 3 and 4 to form and manipulate heterogeneous collections of objects and data values. Two kernel operations provide facilities for constructing heterogeneous sequences and for type-dependent function application.

union: $\text{seq}(\mathbf{OBJ}_1) \times \ldots \times \text{seq}(\mathbf{OBJ}_k) \rightarrow \text{seq}(\mathbf{OBJ}_1 \oplus \ldots \oplus \mathbf{OBJ}_k)$
one_of: $[(\mathbf{OBJ}_1 \rightarrow \mathbf{DATA}_1) \times \ldots \times (\mathbf{OBJ}_k \rightarrow \mathbf{DATA}_k)]$
$\rightarrow (\mathbf{OBJ}_1 \oplus \ldots \oplus \mathbf{OBJ}_k \rightarrow \mathbf{DATA}_1 \oplus \ldots \oplus \mathbf{DATA}_k)$

The **union** operation takes, like **select**, a variable number of operand sequences. It transforms them into a single heterogeneous sequence whose type is the union type of all operands' object types. Each element of any of the operand sequences occurs exactly once in the result sequence. For example, we can easily form the collection of all highways, rivers, and lakes:

Highway River Lake **union**

The **one_of** operation is needed to access the elements of such a heterogeneous sequence. It allows selective application of different functions to different objects of the sequence. **One_of** takes as a parameter a list of functions that map object types into data types and returns a single function from the union object type to the union data type. So in some sense **one_of** is a "function generalization" operator. For example, consider the application of **one_of** to the functions

route:	Highway	→ LINE
flow:	River	→ LINE
surface:	Lake	→ REG

denoted by

one_of[route, flow, surface]

The returned function has functionality Highway ⊕ River ⊕ Lake → LINE ⊕ LINE ⊕ REG. The range type evaluates to EXT. We might use this function as a parameter to **map**:

Highway River Lake **union map**[**one_of**[route, flow, surface]]

After the application of **union** we have a sequence of type seq(Highway ⊕ River ⊕ Lake) which **map** transforms into a sequence of type seq(EXT). **One_of** allows also to select elements from a heterogeneous sequence. We can find all highways, rivers, and lakes in or passing through Germany:

Highway River Lake **union select**[**one_of**[route, flow, surface]

intersects (**the**(State **select**[name ='Germany']) **region**)]

The **select** operation applies the function constructed by **one_of** to each element of the heterogeneous operand sequence. This function in turn selects the "right one" of its component functions and applies

it to the object, namely, to a Highway object the "route" function, and so on. In any case, the result returned is of type EXT and so fits with the functionality of the **intersects** operator.

Note that in general **one_of** can be applied to objects whose type is given by an arbitrary expression built from base object types through the ⊕ and ⊗ constructors. The question arises under which conditions **one_of** is well-defined. First, we observe that with each object function *f*, in general, a large number of functionalities is associated which happens through inheritance, overloading and the construction of product types. That means, when we write

one_of[f_1, …, f_n]

it is not at all clear which of the functions associated with the name f_j is meant. Of course, one might use type annotations, as in:

one_of[name: River → STR, …]

but that appears very uncomfortable in queries. Instead, we disambiguate by considering the object type *s* to which the function constructed by **one_of** is to be applied. Let *dom*(*f*) denote all valid argument types of *f*. For example, we have

dom(pos) = {Location, City, Location ⊗ State, City ⊗ State, …}.

We proceed as follows:

(1) The operand type *s* is brought into disjunctive normal form (as discussed in Section 4) and hence represented as $s_1 \oplus \ldots \oplus s_n$. (The typing of **one_of** requires that the number of functions being arguments for **one_of** and the number of types in the union type match.)

(2) Define the *compatibility relation* as $C = \{(s_i, f_j) \mid s_i \in dom(f_j), 1 \le i, j \le n\}$. Then **one_of** is well-defined if *C* is a (total) function, that is, if each s_i occurs in exactly one of the sets $dom(f_j)$. (Since the number of parameter functions and the number of types in the union are the same *C* is surjective. Moreover, since for each s_i there has to be a parameter function *C* is also injective.)

(3) Now, if **one_of** is to be applied to an object of type s_i, **one_of** chooses the function $C(s_i)$.

Consider the query

Highway (River Lake **union**) **select**[route **intersects one_of**[flow, surface]]

Here the operand type for the result function of **one_of** is Highway ⊗ (River ⊕ Lake). Suppose we had another object type Cube on which "surface" was also defined,

surface: Cube → REAL.

Then the domains of function names "flow" and "surface" would be:

dom(flow) = {River, River ⊗ State, River ⊗ Highway, …}
dom(surface) = {Lake, Cube, Lake ⊗ State, Cube ⊗ State, Lake ⊗ Highway, …}

Step 1. For the example above, the operand type in disjunctive normal form is: (Highway ⊗ River) ⊕ (Highway ⊗ Lake).

Step 2. We observe that Highway ⊗ River ∈ *dom*(flow) and Highway ⊗ Lake ∈ *dom*(surface), so **one_of** is well-defined.

Step 3. The parameter functions of **one_of** are flow: Highway ⊗ River → LINE and surface: Highway ⊗ Lake → REG. Hence it constructs a function of type (Highway ⊗ River) ⊕ (Highway ⊗ Lake) → EXT.

Note that it is possible to do full static type checking at query compile time, so we have to look at the actual objects in the sequence at runtime only to select the right function to apply, and type correctness ensures that we will never miss one.

We can provide the precise types for **union** and **one_of** as done for **select**. For this we need another auxiliary function, *shuffle*, which takes two sequences of equal length, say, $\langle x_1, x_2, \ldots, x_n\rangle$ and $\langle y_1, y_2, \ldots, y_n\rangle$ and a binary type constructor, γ, and produces the sequence $\langle x_1\,\gamma\,y_1, x_2\,\gamma\,y_2, \ldots, x_n\,\gamma\,y_n\rangle$.

union: *fold*(×, *map*(seq, list(**OBJ**)$_i$)) → seq(*fold*(⊕, list(**OBJ**)$_i$))

one_of: *shuffle*(list(**OBJ**)$_i$, list(**DATA**)$_j$, →) → (*fold*(⊕, list(**OBJ**)$_i$) → *fold*(⊕, list(**DATA**)$_j$))

Note that the type for **one_of** could be defined even more flexible: When working on a union of types some which share an overloaded operation symbol, this operation need to be given only once. For example, in the query

City State Highway **union map**[**one_of**[name, hno]]

"name" can be used for objects of type City and State. Unfortunately, this typing is not expressible by multi-level algebra since there is no concept to say that one name represents a set of functions. (Here we have a problem with signatures mixing up syntax and type descriptions: The above signature entry when viewed only as a type specification still covers this extended meaning; read as a syntax specification, however, it requires the same number of parameter functions as the number of types in the union.)

5.3 Graphs

So far, a database could be modeled as an object type hierarchy together with functions applicable to objects, describing their attributes and relationships. We now introduce *graphs* as another modeling tool which means a user can define some part of the database explicitly as a graph structure. Associated with graphs are specific graph operations in the query language algebra such as finding shortest paths, determining certain subgraphs, and so forth. As far as modeling is concerned, explicit graphs and graph operations allow a user to express a query at a very high level. To some extent this is in contrast to graph manipulation in deductive databases where often fairly complex rule programs need

to be written. As far as implementation is concerned, the idea is to provide special storage structures for the representation of graphs and the most efficient graph algorithms available for realizing specific operations. We believe that this approach will result in better performance than can be achieved by optimization of arbitrary rule based programs.

There is a difficulty with our strategy, however: It is not easy to come up with a small collection of graph operations that cover all interesting queries. In that respect, rule-based manipulation, offering more primitive operations that can be combined with a powerful paradigm, has an advantage. Our answer to this problem is that the whole approach needs to be understood within the context of an extensible system: For a given application one should be able to determine the graph operations frequently needed and extend the system by efficient implementations for them. One might also think of adding facilities for rule-based manipulation for those ad-hoc queries that cannot be formulated with the given collection of graph operations, for example, through an algebra operation that takes a collection of rules as a parameter. Still another approach is proposed in [15]: Identify the basic building blocks of a reasonable class of graph algorithms and provide a small set of programming facilities (basically, an iteration operator that can be combined with data structures) so that new algorithms can be expressed in a very compact way. In [15] it is shown how to integrate these ideas into a functional language where the resulting algorithms are as efficient as their imperative counterparts.

In this subsection, we first introduce the part of the type system relevant to graphs. Based on this, we then define a set of operations for graph manipulation. Their use is illustrated by some example queries.

Types

The part of the kind signature relevant to graphs is shown in Figure 7. We have constructors graph, node, edge, xpath, and path with the shown functionalities.

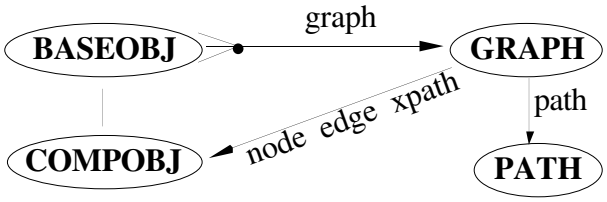

Figure 7

We can pick up any three object sorts *s*, *t*, *u* of kind **BASEOBJ** and form, by application of the graph constructor, a type of graphs over *s*, *t*, *u* denoted by graph(*s*, *t*, *u*). For example, graph(HLocation, Section, Highway) is such a graph type; this sort is of kind **GRAPH**. The constructor semantics of the graph constructor is defined as follows:

$\forall\ s, t, u \in$ «**BASEOBJ**»: $\quad \langle \text{graph}(s, t, u) \rangle = \overline{\text{graph}}(\langle s \rangle, \langle t \rangle, \langle u \rangle) :=$

{ $(N, E, XP, \varepsilon, \pi)$ |

(i) $N \subseteq \langle s \rangle$, $E \subseteq \langle t \rangle$, $XP \subseteq \langle u \rangle$,

(ii) ε: $E \to N \times N$ is total and injective (no two edges between the same nodes),

(iii) π: $XP \to E^*$ is total, its range contains only simple paths of the graph $(N, \varepsilon(E))$.

}

The idea is that in a given graph of type graph(HLocation, Section, Highway) a HLocation object is associated with each node and a Section object with each edge. For each Highway in *XP*, there is a path in this graph associated to it by π.

The constructors node, edge, and xpath are in fact selectors, they extract from a graph sort the sorts it was constructed from. Thus, they can be defined by equations on the type level.

eq node(graph(s, t, u)) = s
edge(graph(s, t, u)) = t
xpath(graph(s, t, u)) = u

Now, the constructor semantics are very simple: $\overline{\text{node}}(\langle \text{graph}(s, t, u) \rangle) := \langle s \rangle$, $\overline{\text{edge}}(\langle \text{graph}(s, t, u) \rangle) := \langle t \rangle$, and $\overline{\text{xpath}}(\langle \text{graph}(s, t, u) \rangle) := \langle u \rangle$. These constructors map to the kind **COMPOBJ** (graph component objects) whose elements can be treated like any other object sort; it is therefore a subkind of **BASEOBJ**. The last constructor is path, defined as:

$\forall$ graph(s, t, u) $\in$ «**GRAPH**»: $\langle \text{path}(\text{graph}(s, t, u)) \rangle = \overline{\text{path}}(\langle \text{graph}(s, t, u) \rangle) :=$
{ $(N, E, XP, \varepsilon, \pi)$ |

(i) – (iii) as in the definition of $\langle \text{graph}(s, t, u) \rangle$

(iv) the graph $(N, \varepsilon(E))$ must be a simple path

}

One recognizes that application of the path constructor just restricts the carrier of a given graph type. Hence for any $G \in$ «**GRAPH**», we have $\langle \text{path}(G) \rangle \subseteq \langle G \rangle$ which allows us to define a subsort relationship in the sort hierarchy:

$\forall$ graph(s, t, u) $\in$ «**GRAPH**»: path(graph(s, t, u)) $\leq$ graph(s, t, u)

In other words, any path can also be viewed as a graph and inherits all operations defined for graphs.

So far, we have just introduced types associated with graphs but no instance yet. We assume that a user can define several object types and then create a graph instance by a data definition command, for example:

create graph Hnet **of type** HLocation, Section, Highway

"Hnet" is now the name of one specific graph in the carrier of graph(HLocation, Section, Highway); right after this command, it is a completely empty graph. We assume that an object type used in such

a graph definition is "devoted to" this graph instance.[5] By that we mean that every object in the object type used for nodes is automatically also a node of this graph instance. When an edge object is created, one has at the same time to specify which nodes it connects. Similarly, when an explicit path object is created, the edges of which it is composed need to be given by the user. Hence for the specific graph Hnet = (N, E, XP, ε, π), we have N = ‹HLocation›, E = ‹Section›, and XP = ‹Highway›. However, the carrier containing Hnet is more general, for example, it also contains all subgraphs of Hnet.

"Hnet" can now be used in a query to refer to one particular (structured) object, just as "River" refers to one particular element in the carrier of seq(River).

Operations

We now introduce a collection of graph operations for use in queries. As explained in the introduction to this subsection, this collection is not meant to be complete. It just illustrates our approach.

op	**subgraph**, **remove**:	$\mathbf{GRAPH}_i \times \text{seq}(\mathbf{NODE}_i)$	$\rightarrow$	$\mathbf{GRAPH}_i$
	subgraph, **remove**:	$\mathbf{GRAPH}_i \times \text{seq}(\mathbf{EDGE}_i)$	$\rightarrow$	$\mathbf{GRAPH}_i$
	nodes:	$\mathbf{GRAPH}_i$	$\rightarrow$	$\text{seq}(\mathbf{NODE}_i)$
	edges:	$\mathbf{GRAPH}_i$	$\rightarrow$	$\text{seq}(\mathbf{EDGE}_i)$
	from, **to**:	$\mathbf{EDGE}_i$	$\rightarrow$	$\mathbf{NODE}_i$
	path:	$\mathbf{XPATH}_i$	$\rightarrow$	$\mathbf{PATH}_i$
	shortest_path:	$\mathbf{GRAPH}_i \times \mathbf{NODE}_i \times \mathbf{NODE}_i \times [\mathbf{EDGE}_i \rightarrow \mathbf{NUM}]$	$\rightarrow$	$\mathbf{PATH}_i$
	circle:	$\mathbf{GRAPH}_i \times \mathbf{NODE}_i \times \mathbf{NUM}_j \times [\mathbf{EDGE}_i \rightarrow \mathbf{NUM}_j]$	$\rightarrow$	$\mathbf{GRAPH}_i$

In this signature specification we use $\mathbf{NODE}_i$, $\mathbf{EDGE}_i$, $\mathbf{XPATH}_i$, and $\mathbf{PATH}_i$ as abbreviations for node($\mathbf{GRAPH}_i$), edge($\mathbf{GRAPH}_i$), xpath($\mathbf{GRAPH}_i$), and path($\mathbf{GRAPH}_i$), respectively. The operations have the following meaning. Given a graph G and a collection of nodes N in it, **subgraph** constructs a subgraph of G consisting of the nodes in N and the edges incident with such nodes; of the explicit paths only those remain whose edges occur in the subgraph. Applied to a graph and a list of edges E, subgraph leaves the nodes alone but in the result graph reduces edges and explicit paths to those in E. **Remove** works in the opposite way: Given a list of nodes it eliminates those nodes as well as all dependent edges and xpaths; given a list of edges, it eliminates these edges and the dependent explicit paths. The operations **nodes** and **edges** return all the nodes or edges, respectively, of a given graph; note that they can also be applied to paths in $\mathbf{PATH}_i$ due to the subsort relationship mentioned above. If these operations are applied to a path, the result sequence contains the nodes or edges in the order of the path. **From** and **to** return the source and target nodes of a given edge. For an explicit path object we can access its path in the graph through the **path** operation which allows to apply path or graph operations to it.

[5] This is not a restriction: If an object type is needed of which only some elements participate in a graph structure, simply define a subtype for those in the graph, as we have done with HLocation and HCity.

Some special graph operations are **shortest_path** and **circle**. Given a graph *G* and nodes *v* and *w* in it, **shortest_path** returns (one of) the shortest path(s) from *v* to *w*. The operation is parameterized by a function assigning a numeric value to an edge. So one can determine shortest paths with respect to the number of edges in a path, the geometric distance or the traveling time in a highway network, or some function combining distance and slope of street sections when planning a bicycle trip. The **circle** operation, applied to a graph *G*, a node *v* in it, and some number *d* determines a subgraph in a "circle of radius *d*" around *v* where the radius is measured in terms of distance on the network which is again determined by a parameter function assigning values to edges. One can, for example, determine a part of a street network that can be reached from a given city within a specified traveling time, or just retrieve all the neighbours of a node in a graph (by assigning 1 to each edge and giving a radius of 1). Note that there are efficient implementations for all of these operations; they are sketched below.

Queries

A highway route from Dortmund to Munich is found by:

fun theCity(*s*:STR):HCity= **the**(HCity **select**[name = *s*])

DoMunich = Hnet theCity('Dortmund') theCity('Munich') **shortest_path**[way **length**]

In order to actually display the path we have to apply a **Show** command with functions returning printable data values, for example,

DoMunich **edges**
Show way

Of course, we can imagine a more sophisticated user interface having available standard display routines for graph/path objects, so that a graph/path object returned by a query would be automatically displayed as is assumed for values of type INT or LINE.

"How far is Munich from Dortmund?"

DoMunich **edges map**[way **length**] **sum**

"Are there any rivers traversed on this trip?"

River **select**[flow **intersects** (DoMunich **edges map**[way] **agg**[**concat**])]

or, alternatively,

River DoMunich **edges select**[flow **intersects** way]

"Which big cities lie on highway 1?"

the(Highway **select**[hno = 1]) **path nodes select**[pop > 500000]

Note that this query works because of the semantics of **select** defined above: First, **path** expands the path associated with the object highway 1 and returns a subgraph (path) of Hnet; then **nodes** returns the list of nodes on this path, of type seq(Location); **select** omits all Location objects for which the function “pop” is undefined, that is, all locations that are not cities.

“Show the part of the highway network that can be reached within half an hour from Hagen!”

Hnet theCity('Hagen') 0.5 **circle**[duration]

Sometimes graph operations are not needed for a particular query about the network: “How many cities in Germany with more than 100000 inhabitants are not on the highway network”:

(**the**(State **select**[name = 'Germany']) cities) **select**[(pop > 100000) **and** (**in**(HCity) **not**)] **count**

Suppose a fog area given by a region (polygon) called “Fog” blocks some part of the highway network. “What is the remaining part of the network?”

Clear = Hnet Section **select**[way **intersects** Fog] **remove**

Extending this query, we can ask: “What route avoiding the fog area can be recommended from Dortmund to Frankfurt with respect to traveling time?”

Clear theCity('Dortmund') theCity('Frankfurt') **shortest_path**[duration]

“How can one get from Dortmund to Düsseldorf if the piece of highway between Bochum and Essen is blocked (say, by an accident)?”

Hnet Section **select**[(**from** name = 'Bochum') **and** (**to** name = 'Essen')] **remove**
theCity('Dortmund') theCity('Düsseldorf') **shortest_path**[duration]

Note that the “name” function is defined only for City objects; we need the extended semantics of **select** mentioned above to allow this short formulation without a type error.

Clearly, the set of graph operations offered is still limited; many interesting queries cannot yet be formulated. The design of a reasonably complete collection of graph operations is a subject of further research. In this paper we just intended to show how graph operations can be integrated into a general querying environment. More operations and example queries can be found in [17].

Implementation of Graph Operations

Let us briefly give an idea about the assumed implementation of the operations offered so far. The **subgraph** and **remove** operations should certainly not produce a copy of the given operand graph. Instead, the sequence given as a second operand is scanned and the corresponding elements of the graph are accessed and marked as valid or invalid for the remaining operations of this query. For example, to implement the query

Hnet Section **select**[way **intersects** Fog] **remove**

the collection of Section objects is accessed, hopefully through a geometric index on the "way" attribute, and a sequence of sections intersecting the fog area is formed. For each of those sections its representation within the graph structure is accessed and marked as invalid for any remaining operations of the query. In [25] it is described how this marking of graph components can be implemented with a time stamp technique.

The **shortest_path** operation is to be implemented by the A* algorithm [36] which is guided by a heuristic function that estimates the distance from a node encountered during the search to the target node.[6] In spatially embedded networks one can often use the Euclidean distance as a very good estimator. Hence a shortest path search will be focused well on the target and run quite fast. An even faster algorithm, which makes use of a geometric index, is described in [14]: For each edge (v, w), the set of nodes to which the shortest path from v leads via w is represented by a polygon with hopefully few bordering edges. Then a shortest path can be reconstructed in linear time (in the number of edges in the path to be found) plus the time to perform point location in the polygons for the successors along the path. Of course, this requires additional storage for the index (ranging from O(n) for Manhattan-like networks to O($n\sqrt{n}$) for most networks found in "real life"). On the other hand it outperforms A*, and the user need not be concerned about heuristics. In general, a **shortest_path** implementation will check the marks of objects encountered and can therefore be restricted to edges marked as valid in this query (by a preceding **subgraph** operation) or avoid edges marked as invalid (by **remove**).

The **circle** operation explores the environment of the given operand node and will be implemented by Dijkstra's single source shortest paths algorithm. The **nodes** and **edges** operations should generally only be applied to a subgraph (in particular, path) obtained as the result of a subquery; if the user applies them to the whole graph, the optimizer should replace this by a reference to the corresponding node or edge object type.

6 Related Work

In the description of our data model we touched quite different areas of database research. So the comparison with other work is divided into several parts: We first look at the underlying functional data model and then discuss the treatment of graphs and heterogeneous collections in other data models. After that, we focus on the specific area of application, spatial data models, and finally, we comment upon the use of multi-level algebra.

Data Modeling in general

The data model described is essentially a functional data model in the spirit of DAPLEX [46]. This

[6] Unfortunately, it is necessary to ask the user to specify such a heuristic function, for example, as
fun distance(x:Location, y:Location):REAL = x pos y pos **mindist**
The function "distance" can then be given as another parameter function to the **shortest_path** operation. This parameter was omitted above to keep the presentation simple. It is unfortunate, because the implementation should in principle not affect the query language, but here in the interest of efficiency it cannot be avoided.

base model is extended by complex objects which can be defined through the use of type constructors. This is similar to FDL [42] or the model presented in [18] in the sense that the model provides a fixed set of type constructors, although the constructors differ significantly: Whereas [42, 18] offer the usual tuple, variant, and sequence constructors we have special sequence, product ($\otimes$), union ($\oplus$), and, of course, graph type construcors. In PROBE [12], complex objects are modeled by the basic facilities of the functional model itself, that is, entity types and functions, and in GENESIS [4] compound structures are represented by (nested) streams. Concerning the query language, FDL provides the full computational power of the λ-calculus. In contrast, we have deliberately chosen a limited functional language since we advocate the use of explicit graphs for the modeling of complex, recursive relationships. In DAPLEX and PROBE, a limited imperative language is used. This imperative style of PROBE is carried over to the language proposed in [43] which meets the traditional view of graph iterations; in [15] it is demonstrated that traversals can also be formulated declaratively and that they can be definitely integrated into a functional language. The supposed lack of completeness concerning types and language of our approach and also of GENESIS must be seen from the viewpoint of extensibility, see below.

The algebraic view of the functional model integrates data model and schema description in a uniform framework. Extending a schema by a new function amounts to simply adding an entry to the signature (and giving a defining expression in case of derived functions). By using the algebraic methodology on a second level the same applies to extensions of the data model (or type system). In GENESIS, structured types are mapped to a collection of streams related by functions (viewed as stream generators), which, as the authors admit, is in some cases not very fortunate. In particular, this applies to the extension by graph structures, where it is suggested to load a graph entirely into main memory before executing queries. Instead, we propose to provide special storage structures for newly added types, such as graphs [25]. Certainly, this approach requires more implementation efforts but results in higher performance.

From the semantic data modeling point of view, the functional data model provides objects, attributes, type hierarchies and derived data. Aggregation is possible by associating to an entity each item to be aggregated by a function (or by the explicit use of type constructors as in FDL). Association is realized in DAPLEX by multi-valued functions; we use the seq constructor, which must not be nested as is possible in FDL or GENESIS. In particular, some of the essential features of object-oriented languages, such as object identity, subtyping, and inheritance are covered by the functional data model.

Graphs in Databases

Each data model has its own facilities to represent relationships among objects. Since graphs are a special concept for representing such relationships many data models do not worry about graphs at all. There are, however, at least two reasons for considering graphs additionally (or instead): The first

is, that many “real-life” problems can be directly expressed in terms of graph concepts (paths, spanning trees, matchings, and so on) that are well understood and thus easy to deal with (compared with reinventing representations in special data model structures). The second observation is that for most of these problems efficient algorithms are available which can be used if the application problem is formulated as a graph problem.

A first step for taking graphs into account is to regard certain structures of the data model as a graph (in the relational model, for example, a relation with two attributes, say, “from” and “to”) and define graph operations which are only applicable to such structures. Concerning the relational data model, this line is followed by [2] defining a general transitive closure operator and by [6] describing an extension of SQL. In [43] a method for traversing graphs is proposed in the context of the functional data model.

Proposals to use graphs directly for data modeling and to define query languages in terms of graphs are made by [10, 21]. In the work of Cruz et al. the intention is to focus on the underlying graph structure of a database and also to employ efficient graph algorithms [11]. In [21] the main purpose is to use graphs as a unifying concept for the different parts of data models. In these proposals *only* graphs are visible for data modeling and querying. In contrast, our approach is to offer graphs as a separate, “first-class” concept besides other facilities of the functional model. An approach for viewing graphs as an additional feature in a relational system is sketched in [25].

Heterogeneous Collections

In O_2 [9] it is possible to form the union of types *s* and *t* if there is an appropriate schema definition relating *s* and *t* as (possibly indirect) subtypes of another type *u*. Assuming that *u* is the smallest common supertype of *s* and *t*, the result of the union is of type *u* which means that only attributes defined on *u* may be applied in later steps of the query. In Machiavelli [7, 39] it is possible to form the union of types *s* and *t* even if there is no schema definition as described above for O_2. The types of *s* and *t* are assumed to be tuples of data types. The same holds for the model presented in [25] where the data types are additionally arranged into a hierarchy and the result type is obtained by taking least upper bounds with respect to the data type hierarchy for any two matching tuple components in the types *s* and *t*. However, attributes that do not match cannot be accessed any more after forming the union. In the model presented here it is possible to form the union of types *s* and *t* with no restriction and all attributes are preserved for later use in the query. This is also possible in FAD [3]. However, in FAD the union of two types *s* and *t* is simply of type *set*. Attributes can selectively be accessed by means of an *if-then-else* function. But FAD is not strongly typed and erroneous attempts to access attributes in such a set cannot always be ruled out. Thus, our proposal seems to be the most general treatment of heterogeneous collections allowing for static type checking.

Data Models for Spatial Databases

In the development of spatial database systems, on the modeling side the main focus has been on the

definition of spatial data types and operations and their integration as “abstract data types” into database models. Most often, the relational model was used as a basis (for example, [19, 44, 28, 13]), but there are also approaches based on the functional model [34, 40] or on an extended ER-model [31]. Another important issue has been the handling of partitions of the plane, sometimes called maps, including map operations such as overlay [35, 45]. The need of treating graphs in geographic applications has also been observed in [33].

Concerning heterogeneous collections, current prototype systems and query language proposals offer only superficial support for this such as syntactic constructs to formulate the request in a single query [1, 19] and output facilities to overlay the results obtained from the different object classes. At the implementation level, the query is decomposed into distinct queries on all involved object classes which results in a lot of overhead. Generally, there are no clean modeling concepts to deal with such heterogeneous collections of objects. The model of [31] provides a partial solution by organizing spatial object classes into a generalization hierarchy, but one cannot query together object classes unrelated in the hierarchy.

Multi-Level Algebra

In this paper, we have used multi-level algebra for the description of our data model. Multi-level algebra itself is described and discussed thoroughly in [16]. In particular, its use for specifying various aspects of type systems is illustrated by many examples, including a specification of the relational model and an NF^2 model.

In fact, two-level algebras were already used in [41] to specify categories with certain properties for theoretical investigation and in [30] for the formalization of the composition of specifications. In contrast, our concern is the formal description of a concrete data model. In that respect the approach presented in [26] is very similar, although directed more towards describing different levels of a system architecture. Unlike [41, 30, 26] our approach is not limited to two levels, and it is shown in [16] that especially a third level can be extremely helpful: One usage is to describe overloading of operations with functions on different numbers of parameters (which is needed in our model, for example, to formally describe the **select** or **union** operation).

7 Conclusions

Within the formal framework of a multi-level order-sorted algebra we have presented a data model that employs functional modeling concepts to describe properties of objects and relationships between objects. The main purpose in developing the model was the integration of graphs and graph operations into a general modeling and querying environment. In contrast to earlier work such as [2, 43] graphs are not modeled through the standard facilities of a given data model (for example, relations of a special form) but explicitly. In contrast to work focusing on the graph structure and graph operations as

such, like [10, 21], graphs are not the only tool, but are part of a more general environment. We have shown that explicit graphs facilitate the clear and direct modeling of graph structures and that queries can be formulated in a very intuitive way. At the same time, by choosing appropriate operations, efficient implementations can be provided.

Through the introduction of union type constructors on data and object types together with **union** and **one_of** operations in the query language a flexible treatment of heterogeneous collections of objects was achieved within the realm of static type checking.

Some details in the model deserve to be highlighted. The view of kernel operations, for example, **select**, **asc**, as higher order functions offers a clean formalization; this is an improvement over other approaches, for example "parameter expressions" in the NST-algebra [27]. The multi-level order-sorted algebra in connection with the seq constructor and higher order functions makes it possible to define operations in the most general and powerful way, for example, to capture the essence of sequence operations by making them applicable to seq(**ANY**). For instance, the **asc/desc** operators can sort any kind of sequence by the values of functions applicable to the elements of the sequence. We have introduced operators with a variable number of operands. This is generally useful when operators are needed in a query language that are originally binary and associative (join, cartesian product, union). The well-known problem in algebraic query languages that the order of operations is over-specified (in particular, join order) can so be eliminated. The **select** operator that we have introduced can be used as comfortably as the select-from-where construct in SQL dialects; it does not specify join order nor whether a selection, join or cartesian product is desired.

An implementation of at least parts of the model presented here is underway using the extensible database system Gral [24, 5] as a basis. A special storage structure for graphs decribed in [25] has already been implemented; it clusters and links node, edge, and explicit path objects in such a way that connections between any of these objects can be followed efficiently. For the implementation of graph operations efficient graph algorithms, as indicated in Section 6.3, are to be used; here A* has so far been realized. The functional, or algebraic, structure of the query language is particularly suitable for optimization, a rule-based paradigm as in [32, 5] can be used, and the special optimization opportunities offered by the functional data model can be exploited [18] (in particular, access path optimization can be carried out already on the algebraic level).

Acknowledgements

We would like to thank the referees for their detailed comments and suggestions that helped to improve the paper.

References

[1] D.J. Abel: SIRO-DBMS: A Database Tool Kit for Geographical Information Systems, *Int. Journal of Geographical Information Systems 3*, 1989, pp. 103-116.

[2] R. Agrawal: Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries, *3rd IEEE Int. Conf. on Data Engineering*, 1987, pp. 580-590.

[3] F. Bancilhon, T. Briggs, S. Khoshafian & P. Valduriez: FAD, a Powerful and Simple Database Language, *13th Int. Conf. on Very Large Data Bases*, 1987, pp. 97-105.

[4] D.S. Batory, T.Y. Leung & T.E. Wise: Implementation Concepts for an Extensible Data Model and Data Language, *ACM Transactions on Database Systems Vol. 13*, No. 3, 1988, pp. 231-262.

[5] L. Becker & R.H. Güting: Rule-Based Optimization and Query Processing in an Extensible Geometric Database System, *ACM Transactions on Database Systems Vol. 17*, No. 2, 1992, pp. 247-303.

[6] J. Biskup, U. Räsch, & H. Stiefeling: An Extension of SQL for Querying Graph Relations, *Computer Languages 15*, 1990, pp. 65-82.

[7] V. Breazu-Tannen, P. Buneman, & A. Ohori: Static Type Checking in Object-Oriented Databases, *Data Engineering Vol. 12*, No. 3, 1989, pp. 5-12.

[8] L. Cardelli: Types for Data Oriented Languages, *Int. Conf. on Extending Database Technology*, 1988, pp. 1-15.

[9] S. Cluet, C. Delobel, C. Lécluse & P. Richard: Reloop, an Algebra Based Query Language for an Object-Oriented Database System, *1st Int. Conf. on Deductive and Object-Oriented Databases*, 1989, pp. 294-313.

[10] I.F. Cruz, A.O. Mendelzon & P.T. Wood: G^+: Recursive Queries Without Recursion, *2nd Int. Conf. on Expert Database Systems*, 1988, pp. 645-666.

[11] I.F. Cruz, & T.S. Norvell: Aggregative Closure: An Extension of Transitive Closure, *5th IEEE Int. Conf. on Data Engineering*, 1989, pp. 384-391.

[12] U. Dayal & J.M. Smith: PROBE: A Knwoledge-Oriented Database Management System, in M.L. Brodie & J. Mylopoulos (eds.): *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Springer-Verlag, 1986.

[13] M.J. Egenhofer: Spatial SQL: A Query and Presentation Language, Technical Report 103, Department of Surveying Engineering, University of Maine, 1989, to appear in: *IEEE Transactions on Knowledge and Data Engineering*.

[14] M. Erwig: Encoding Shortest Paths in Spatial Networks, Report 116, FernUniversität Hagen, 1991.

[15] M. Erwig: Graph Algorithms = Iteration + Data Structures ? The Structure of Graph Algorithms and a Corresponding Style of Programming, *18th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 657, 1992, pp. 277–292.

[16] M. Erwig: Specifying Type Systems with Multi-Level Order-Sorted Algebra, *3rd Int. Conf. on Algebraic Methodology and Software Technology*, 1993, pp. 179-188.

[17] M. Erwig & R.H. Güting: Explicit Graphs in a Functional Model for Spatial Databases, Report 110, FernUniversität Hagen, 1991. Revised Version 1993.

[18] M. Erwig & U.W. Lipeck: A Functional DBPL Revealing High Level Optimizations, *3rd Int. Workshop on Database Programming Languages*, 1991, pp. 306–321.

[19] A. Frank: MAPQUERY: Data Base Query Language for Retrieval of Geometric Data and their Graphical Representation, *Computer Graphics 16*, 1982, pp. 199-207.

[20] J.A. Goguen & J. Meseguer: Order-Sorted Algebra I: Partial and Overloaded Operations, Errors and Inheritance, Report, SRI International, 1989.

[21] M. Gyssens, J. Paredaens & D. van Gucht: A Graph-Oriented Object Database Model, *ACM Conf. on Principles of Database Systems*, 1990, pp. 417-424.

[22] R.H. Güting: Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems, *Int. Conf. on Extending Database Technology*, 1988, pp. 506-527.

[23] R.H. Güting: Modeling Non-Standard Database Systems by Many-Sorted Algebras, Technical Report 255, Fachbereich Informatik, Universität Dortmund, 1988.

[24] R.H. Güting: Gral: An Extensible Relational Database System for Geometric Applications, *15th Int. Conf. on Very Large Data Bases*, 1989, pp. 33-44.

[25] R.H. Güting: Extending a Spatial Database System by Graphs and Object Class Hierarchies, in G. Gambosi, H. Six, and M. Scholl (eds.): *Int. Workshop on Database Management Systems for Geographical Applications*, Capri, 1991.

[26] R.H. Güting: Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization, *ACM SIGMOD Conf. on Management of Data*, 1993, pp. 277-286.

[27] R.H. Güting, R. Zicari & D.M. Choy: An Algebra for Structured Office Documents, *ACM Transactions on Office Information Systems Vol. 7*, No. 4, 1989, pp. 123-157.

[28] T. Joseph & A. Cardenas: PICQUERY: A High Level Query Language for Pictorial Database Management, *IEEE Transactions on Software Engineering 14*, 1988, pp. 630-638.

[29] S. Kaes: Type Inference in the Presence of Overloading, Subtyping and Recursive Types, *ACM Conf. on Lisp and Functional Programming*, 1992, pp. 193-204.

[30] J. Leszczylowski & M. Wirsing: Polymorphism, Parameterization and Typing: An Algebraic Specification Perspective, *Symp. on Theoretical Aspects of Computer Science*, 1991, pp. 1-15.

[31] U. Lipeck & K. Neumann: Modelling and Manipulating Objects in Geoscientific Databases, *5th Int. Conf. on the Entity-Relationship Approach*, 1987, pp. 67-86.

[32] G.M. Lohman: Grammar-like Functional Rules for Representing Query Optimization Alternatives, *ACM SIGMOD Conf. on Management of Data*, 1988, pp. 18-27.

[33] M. Mainguenaud: GROG: Geographical Queries Using Graphs, *Advanced Database System Symposium*, 1989.

[34] F. Manola, & J.A. Orenstein: Toward a General Spatial Data Model for an Object-Oriented DBMS, *12th Int. Conf. on Very Large Data Bases*, 1986, pp. 328-335.

[35] P.E. Mantey & E.D. Carlson: Integrated Geographic Data Bases: The GADS Experience, in: A. Blaser (ed.) *Data Base Techniques for Pictorial Applications*, Springer, 1980, pp. 173-198.

[36] N.J. Nilsson: *Principles of Artificial Intelligence*, Springer, 1982.

[37] T. Nipkow & C. Prehofer: Type Checking Type Classes, *20th Symp. on Principles of Programming Languages*, 1993, pp. 409-418.

[38] T. Nipkow & G. Snelting: Type Classes and Overloading Resolution via Order-Sorted Unification. *Conf. on Funct. Programming and Computer Architecture*, LNCS 523, 1991, pp. 1-14.

[39] A. Ohori, P. Buneman & V. Breazu-Tannen: Database Programming in Machiavelli – a Polymorphic Language with Static Type Inference, *ACM SIGMOD Conf. on Management of Data*, 1989, pp. 46-57.

[40] J.A. Orenstein & F. Manola: PROBE: Spatial Data Modeling and Query Processing in an Image Database Application, *IEEE Transactions on Software Engineering 14*, 1988, pp. 611-629.

[41] A. Poigné: On Specifications, Theories, and Models with Higher Types, *Information and Control 68*, 1986, pp. 1-46.

[42] A. Poulovassilis & P. King: Extending the Functional Data Model to Computational Completeness, *Int. Conf. on Extending Database Technology*, 1990, pp. 75-91.

[43] A. Rosenthal, S. Heiler, U. Dayal & F. Manola: Traversal Recursion: A Practical Approach to Supporting Recursive Applications, *ACM SIGMOD Conf. on Management of Data*, 1986, pp. 166-176.

[44] N. Rossopoulos, F. Faloutsos & T. Sellis: An Efficient Pictorial Database System for PSQL, *IEEE Transactions on Software Engineering 14*, 1988, pp. 639-650.

[45] M. Scholl & A. Voisard: Thematic Map Modeling, *1st Int. Symp. on Large Spatial Databases*, 1989, pp. 167-190.

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