The GraphDB Algebra: Specification of Advanced Data Models with Second-Order Signature

Ludger Becker

Westfälische Wilhelms-Universität
FB 15 - Informatik, Einsteinstr. 62
D-48149 Münster
GERMANY
beckelu@math.uni-muenster.de

Ralf Hartmut Güting

Praktische Informatik IV
Fernuniversität Hagen
D-58084 Hagen
GERMANY
gueting@fernuni-hagen.de

Abstract: A framework using so called *second-order signature* for the specification of database models has been presented in earlier work. The goal of this approach is to provide generic tools for the implementation of database systems, in particular for parsing and rule-based optimization and for execution of query plans, that can be used with widely varying data models and query languages. In this paper we apply this specification technique to the graph based data model *GraphDB*. We develop an algebraic description for the querying facilities of GraphDB and use second-order signature to specify the GraphDB data model and its algebra.

Keywords: Extensible databases, system architecture, specification, type systems, algebra, modeling, graph databases, second-order signature.

1 Introduction

Extensible database systems have been studied for more than a decade. Today these systems support extensions at all levels of the system. We may add representations of types, procedures for operations, special types of index structures, new query processing methods, and extensions to the optimizer. The extensible optimizer is required to map operations of the query language to efficient operations on the underlying index structures and query processing algorithms. Quite a few extensible systems have been built and on the engineering side a lot of progress has been made (e.g. [GrDe 87, Haas 90, SPSW 90, SRH 90, GrMc 93, Grae 94]). However, most systems lack a formal framework to define what extensibility means. What kind of data models are supported? Which additions to representation structures and query processing are possible?

The goal of our work is to provide a clean extensible architecture based on a precise formal framework. This architecture is shown in Figure 1.

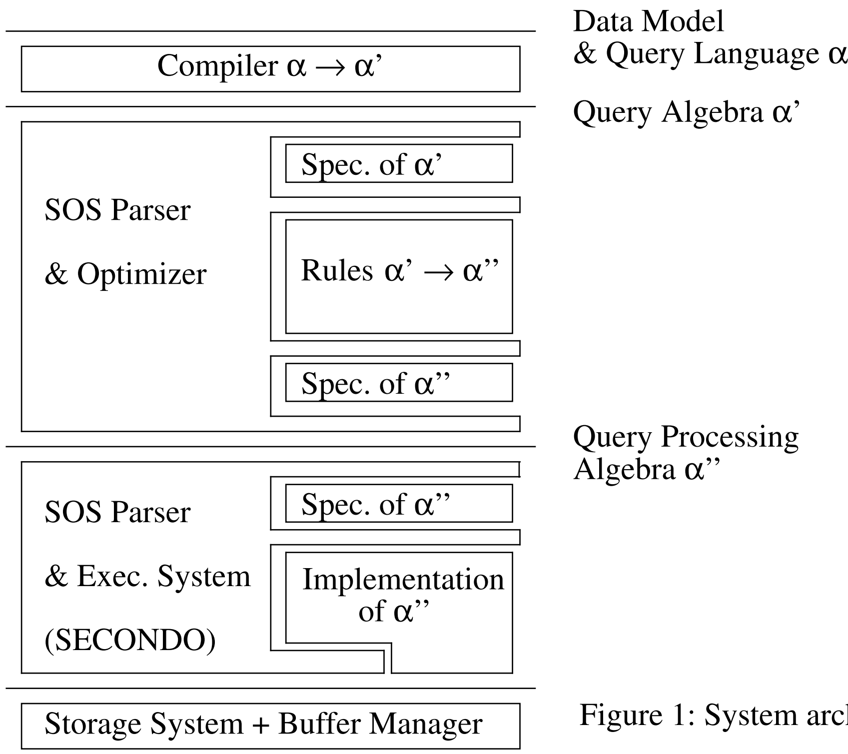

Figure 1: System architecture

Under this architecture, to implement a new data model and query language α, one should design a query algebra α' and a query processing algebra α''. A relatively simple compiler $\alpha \rightarrow \alpha'$ has to be written (probably with help of a parser generator tool like *yacc*) to translate updates and queries in the α DDL and DML into corresponding operations of algebra α'. Below that level, two powerful tools take over. The first is a general parser & optimizer component (never mind the SOS for the moment) which takes a specification of the query algebra α', another specification of the query processing algebra α'', and a (structured) collection of rules describing transformations from α' to α''. The optimizer will then be able to translate queries formulated in α' into query plans in α''. The second tool is a general parser & execution system which takes a specification of the query processing algebra α'' and an implementation of α'' in the form of a set of data structures (for the sorts of the algebra) and a set of procedures (to realize the operations of the algebra). The execution system will then be able to execute a query plan written in the algebra α''. Note that both tools, the optimizer as well as the execution system, are completely independent from the data model α, and can therefore serve to implement a wide variety of database systems.

Clearly, to make this architecture feasible, it is crucial to have a formal specification framework which allows one to describe precisely widely varying data models and query languages (query algebras) as well as representation models and query processing algebras. This is because the specification framework is the basis for the implementation of the generic optimizer and execution system tools which will read these specifications to implement particular database systems.

Such a specification framework, called *second-order signature* (SOS), was proposed in [Güti 93]. The basic idea is to use a system of two coupled signatures where the first signature describes a type system and the second one an algebra over the types of the first signature. The type system can describe either the data model or the representation structures of a system. The algebra can either describe querying at a conceptual level or query processing. The idea was applied in [Güti 93] to describe the relational model and algebra and stream based query processing in an extended relational database system. An open question was whether the framework provides sufficient "expressive power" for the description of more complex data models such as object-oriented or graph-based models and their respective query processing systems, or which extensions to the method were needed.

In this paper, we "test" the specification method on an advanced data model, the object-oriented and graph-based model *GraphDB* [Güti 94a, Güti 94b], by designing a query algebra for GraphDB within the SOS framework. The GraphDB model integrates data modeling for traditional applications with the modeling and querying of network or graph structures (e.g. highway networks as an example of spatial networks). It also offers some key features of object-oriented data models, e.g. objects having identity and tuple structure, attributes which may be data or object-valued, and classes organized in an inheritance hierarchy. To describe a network structure the model distinguishes simple and link objects, defining the nodes and edges of a graph, and path objects, describing explicitly stored paths in this graph. An algebra for such a model is quite interesting in its own right.

The main contributions of this paper are the following:

- We demonstrate the feasibility of the second-order specification framework to describe advanced data models and show a number of specification techniques that can be used for such models.
- A few powerful extensions to the method as presented in [Güti 93] have been discovered in the design of the GraphDB algebra. The most important is the introduction of user-defined predicates (in the style of logic programming) which provide general programming capability for type checking and computation of result types in specifications.
- The GraphDB algebra shows typing and algebraic modeling for a number of graph-based concepts which have not yet been captured in object-oriented query algebras (e.g. [ShZd 90,VaDe 91]), for example for path types, description of relevant subgraphs in queries, or generation of link objects in queries.

The paper is organized as follows: In Section 2 we introduce the basic concepts of second-order signature. Section 3 introduces an SOS specification for the type system of GraphDB, and Section 4 introduces additional types and operators to model querying in GraphDB. Section 5 describes related work, and Section 6 concludes the paper.

2 Specification of Data Models

In this section, we review *second order signature*, introduced in [Güti 93] as a tool for the specification of data models. First, some well-known definitions of signature, terms, etc. are recalled. We then explain the basic idea of second-order signature and show how it can be used to define a type system for the relational model and some of the relational operators. This paper introduces a few extensions to the specification techniques of [Güti 93]. For a formal definition of second-order signature see [Güti 93].

2.1 Signatures

Second order signatures extend the concept of a *signature*, which is well known from the specification of abstract data types. Signatures consist of a set of sorts and operator symbols:

Definition (signature) A *signature* is a pair (S, Σ), where

- S is a set (whose elements are called *sorts*).
- $\Sigma = \{\Sigma_{w,\, s}\}_{w \in S^*,\, s \in S}$, is a family of sets (whose elements are called *operators*).

A signature has an associated set of terms:

Definition (terms) Let (S, Σ) be a signature. The set $T_{\Sigma\, s}$ of *terms of sort s* is defined as follows:

(1) A constant $\omega\colon \to s$ is a term of sort s.

(2) If $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ and $\omega\colon s_1 \times \ldots \times s_n \to s$ is an operator, then $\omega\ (t_1, \ldots, t_n)$ is a term of sort s.

T_Σ denotes the S-indexed set $\{T_{\Sigma\, s}\}_{s \in S}$ of terms over Σ.

The semantics of a signature is defined by an algebra consisting of a (carrier) set for each sort in the signature and a function on these sets for each operator in the signature. These functions must have domains and range according to the string of sorts of the operator:

Definition (algebra) Let Σ be an S-sorted signature. An (S, Σ)-*algebra* $A = (S_A, \Omega_A)$ is defined by:

- $S_A = \{s_A\}_{s \in S}$ where each s_A is a set (called the carrier of s).
- $\Omega_A = \{\omega_A : s_{1,\, A} \times \ldots \times s_{n,\, A} \to s_A \mid \omega \in \Sigma_{w,\, s} \text{ for } w = s_1 \ldots s_n\}_{w \in S^*,\, s \in S}$, where each ω_A is a function with the indicated domain and range sets in S_A.

This concept of signature is now first extended to make for a given signature also "automatically" list sorts, product sorts, union sorts, and function sorts available.

Definition (extended signature) Given a set of sorts S, an *extended S-sorted signature* (*e-signature*) Σ is a signature $(\overline{S}, \Sigma)$, where $\overline{S}$ is defined as follows:

(1) $s \in S \Rightarrow s \in \overline{S}$

(2) If for $n \geq 2$ $s_1, \ldots, s_n$ are sorts in $\overline{S}$, then $(s_1 \times \ldots \times s_n)$ is a sort in $\overline{S}$.

(3) If for $n \geq 2$ $s_1, \ldots, s_n$ are sorts in $\overline{S}$, then $(s_1 \cup \ldots \cup s_n)$ is a sort in $\overline{S}$.

(4) If $s \in \overline{S}$, then s^+ is a sort in $\overline{S}$.

(5) If $s \in \overline{S}$, then s^* is a sort in $\overline{S}$.

(6) If for $n \geq 0$ $s_1, \ldots, s_n$ and s are sorts in $\overline{S}$, then $(s_1 \times \ldots \times s_n \to s)$ is a sort in $\overline{S}$.

Extended signatures have an extended set of terms:

Definition (terms of an S-sorted e-signature) For an S-sorted e-signature Σ the set $\overline{T}_{\Sigma\, s}$ of *terms of sort s* is defined as follows:

(1) A constant $\omega\colon \to s$ is a term of sort s. If $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$ and $\omega\colon s_1 \times \ldots \times s_n \to s$ is an operator, then $\omega\ (t_1, \ldots, t_n)$ is a term of sort s.

(2) If $t_1, \ldots, t_n$ are terms of sorts $s_1, \ldots, s_n$, then $(t_1, \ldots, t_n)$ is a term of sort $(s_1 \times \ldots \times s_n)$.

(3) If t is a term of sort s_1 or of sort s_2 … or of sort s_n, then t is a term of sort $(s_1 \cup \ldots \cup s_n)$.

(4) If for $n \geq 1$ $t_1, \ldots, t_n$ are terms of sort s, then $<t_1, \ldots, t_n>$ is a term of sort s^+.
(5) If for $n \geq 0$ $t_1, \ldots, t_n$ are terms of sort s, then $<t_1, \ldots, t_n>$ is a term of sort s^*.
(6) If for $n \geq 0$ $x_1, \ldots, x_n$ are variables of sorts $s_1, \ldots, s_n$, and t is a term of sort s with free variables $x_1, \ldots, x_n$, then **fun**$(x_1\colon s_1, \ldots, x_n\colon s_n)\ t$ is a term of sort $(s_1 \times \ldots \times s_n \rightarrow s)$. Also, if for $n \geq 1$, $\omega\colon s_1 \times \ldots \times s_n \rightarrow s$ is an operator, then ω is a term of sort $(s_1 \times \ldots \times s_n \rightarrow s)$.

For example, suppose *int* is a sort and $0\colon \rightarrow int$ an operator of a given signature (S, Σ). Then e.g. $(int \times int)$, int^*, and $(int \rightarrow int)$ are sorts of the extended signature, and (0, 0), $<0, 0, 0, 0>$, and **fun**(x: *int*) x are corresponding terms.

This formal definition requires prefix syntax for the application of operators (e.g. $+(x, y)$ for an addition operator). This is relaxed to make expressions (queries) more readable; it is possible to specify a *syntax pattern* for each operator (see Section 2.3).

The basic idea of second-order signature is now to use a system of two coupled (extended) signatures to describe a data model. The first signature defines a *type system*; here the sorts of the signature describe so-called *kinds* and the operators *type constructors*. Terms of this signature describe the available *types* of this type system. The second signature uses the types of the first signature as sorts and defines operators (an algebra) over these types. In particular, since types are classified by their result kind (just as terms are classified by their result sort), one can easily specify polymorphic operators by quantification over kinds. We shall now illustrate this by giving example specifications for the relational model and algebra.

2.2 Specification of a Type System

In this section we specify a type system for the relational model, i.e., we have to define a set of kinds and a set of type constructors. The kinds are IDENT, DATA, TUPLE, and REL. IDENT offers a type for identifiers (used for attribute names), DATA are atomic data types, TUPLE denotes tuple types, and REL contains types of relations.

	$\rightarrow$ IDENT	$\underline{ident}$
	$\rightarrow$ DATA	$\underline{integer}$, $\underline{real}$, $\underline{string}$, $\underline{bool}$
$\forall$ *list* **in** $(\underline{ident} \times \text{DATA})^+$, *noDuplicateNames*(*list*):		
list	$\rightarrow$ TUPLE	$\underline{tuple}$
TUPLE	$\rightarrow$ REL	$\underline{rel}$

There is exactly one type of kind IDENT and there are four types of kind DATA. In contrast, there is an infinite number of types of the kinds TUPLE and REL. For any given tuple type t in the kind TUPLE, $\underline{rel}(t)$ is a corresponding relation type (schema) in kind REL. The description of the $\underline{tuple}$ type constructor is a bit more complex and already introduces some specification techniques that will be used for defining type constructors as well as operators of an algebra. We may introduce variables denoting terms. For type constructors, the sort of these terms can be built from kinds *and from types* (this is discussed below). In the specification above *list* denotes a list of pairs where the first component of each pair is an identifier (a value of type $\underline{ident}$) and the second component is an atomic data type (a type of kind DATA). The possible bindings are constrained by a predicate *noDuplicateNames*. This predicate checks whether all attribute names for a binding of *list* are distinct. For the sake of simplicity we do not present the logical programs used to define the predicates of a specification in this paper. The specification above first defines a type constructor $\underline{tuple}$ taking an operand of sort $(\underline{ident} \times \text{DATA})^+$. Second for all bindings of *list* satisfying the predicate *noDuplicateNames* *tuple*(list) is a type of kind TUPLE. Hence, for example,

tuple(<(name, *string*), (age, *integer*)>)   is a type of kind TUPLE, and
rel(*tuple*(<(name, *string*), (age, *integer*)>))   is a type of kind REL.

The fact that type constructors have kinds as well as types as domains (that is, take types as well as values of types as arguments) is a bit confusing at first but is crucial for many specifications. In fact, it is quite natural. For example, in programming languages we may have an array definition of the form "**array** [100] **of** integer", where **array** is a type constructor taking type *integer* as well as the value 100 as arguments.

It is obvious that type constructor specifications can have cycles, since we may use types to define variables in specifications. Such cyclic specifications must be avoided.

2.3 Specification of Operators

Once a type system has been defined, one can specify operations on these types, using quantification over kinds. As an example we consider some operations for the relational model:

∀ *data* **in** DATA:
    data × *data*   → *bool*   <, >, =, ≠, , ≤, ≥   _#_

Since *data* ranges over the types of kind DATA, this specification defines comparison operators <, >, =, ≠, , ≤, ≥ for each atomic type (*string*, *integer*, *real*, *bool*). The specification also includes the syntax of an application of the operators (_#_). The operator symbol denoted by # is enclosed by the two operands denoted by _. Next we consider the relational selection:

∀ *rel*: *rel*(*tuple*) **in** REL:
    rel × (*tuple* → *bool*)   → *rel*   **select**   _#[_]

The variable *rel* and the variable *tuple* are bound by the variable definition. *rel* denotes any relation type and *tuple* denotes the *tuple* type corresponding to this relation type. The **select** operator takes two operands: a relation of type *rel* and a mapping from the tuple type *tuple* underlying *rel* to *bool*. The syntax is described by the pattern _#[_] specifying the order "first operand, operator, second operand in square brackets". This specification defines the operand types and the result type of **select**. It does not say how the result of an application of **select** is determined. This is only fixed when an algebra is associated with the signature.

The next specification describes the access to attribute values which is required for defining selection predicates:

∀ *tuple* **in** TUPLE, ∀ *name* **in** *ident*, ∀ *dtype* **in** DATA, *member*(*name*, *dtype*, *tuple*):
    tuple   → *dtype*   *name*   _#

This specification introduces three variables, a variable *tuple* denoting a tuple type, a variable *name* denoting an identifier, and a variable *dtype* describing an atomic type. The predicate *member* ensures that *name* and *dtype* denote an attribute of the tuple type *tuple*. The specification defines an operator for each identifier of the carrier set corresponding to *ident*. If the *member*-predicate is satisfied, the application of the operator denoted by *name* to a tuple of type *tuple* is valid. The result of applying the operator is an atomic value of the type denoted by *dtype*.

Combining the operators defined so far, we can formulate a query (assuming *people* is an object of the relation type mentioned above and *person* is the corresponding tuple type):

query people **select**[**fun**(p: person) (p age) > 50]

The **join** operator can be specified as follows:

$\forall$ rel_1: $\underline{rel}(tuple_1)$ **in** REL, $\forall$ rel_2: $\underline{rel}(tuple_2)$ **in** REL, $\forall$ rel_3: $\underline{rel}(tuple_3)$ **in** REL,
$concat(tuple_1, tuple_2, tuple_3)$:
$rel_1 \times rel_2 \times (tuple_1 \times tuple_2 \rightarrow \underline{bool})$ $\rightarrow rel_3$ **join** _ _#[_]

This specification introduces three variables denoting relation types and three variables denoting the corresponding tuple types. The third type is the result type of the join operation. Similar to the **select** operator, the join predicate is described by a mapping from the pair of tuple types corresponding to the two operand relations to the type *<u>bool</u>*. The tuple type corresponding to the result relation is described by the predicate *concat*. This predicate ensures that the attribute list corresponding to the result relation is the concatenation of the tuple types corresponding to the two operand relations.

As usual in logic programming, there is a declarative as well as a procedural interpretation of predicates such as *concat*. The declarative interpretation has just been stated. Under the procedural interpretation, *concat* forms the concatenation of lists $tuple_1$ and $tuple_2$ to construct the list $tuple_3$. Hence, the purpose of this predicate is actually to construct the result type of the join.

To make sure that a given specification works (that is, type checking and computation of result types is possible), one needs to check whether all variables in the specification can be bound starting from a given operator application. It is instructive to check this for the **join** operator. Here an application would be:

rel_1 rel_2 **join**[**fun**(s: $tuple_1$, t: $tuple_2$) *expr*]

From the call, the variables rel_1, rel_2, $tuple_1$, and $tuple_2$ are bound already. In fact, an implementation of an SOS parser may even allow a short form such as

rel_1 rel_2 **join**[**fun**(s, t) *expr*]

Now rel_1 binds $tuple_1$ through the specification $\forall$ rel_1: $\underline{rel}(tuple_1)$ **in** REL; similarly, rel_2 binds $tuple_2$. Then $tuple_1$ and $tuple_2$ bind $tuple_3$ via the *concat* predicate (which is surely able to compute the concatenation of two given lists). Finally, $tuple_3$ binds rel_3 through the specification $\forall$ rel_3: $\underline{rel}(tuple_3)$ **in** REL.

As a further example, let us check the bindings in the attribute access specification:

$\forall$ *tuple* **in** TUPLE, $\forall$ *name* **in** *<u>ident</u>*, $\forall$ *dtype* **in** DATA, *member*(*name*, *dtype*, *tuple*):
tuple $\rightarrow$ *dtype* *name* _#

A call has the form:

tuple name

Hence, *tuple* and *name* are bound by the call and they bind *dtype* through the *member* predicate which only has to check whether *name* occurs as a first component of one of the pairs of list *tuple* and bind *dtype* to the corresponding second component.

Instead of predicates, an alternative technique for describing the construction of complex new result types (as for **join** above) is the use of special type mapping functions called *type operators*. Using this technique, the specification of the join operator looks as follows:

$\forall\ rel_1$: $\underline{rel}(tuple_1)$ **in** REL, $\forall\ rel_2$: $\underline{rel}(tuple_2)$ **in** REL:
$rel_1 \times rel_2 \times (tuple_1 \times tuple_2 \to \underline{bool}) \quad \to rel$: REL **join** _ _#[_]

Here the description of the result type "*rel*: REL" should be read as "some type *rel* in REL" to be determined by a type operator for **join** which we denote as τ_{join}. A function definition for τ_{join} has to be supplied as a part of a second-order algebra which is the semantics of an SOS specification [Güti 93]. Such a type mapping function is called during the parsing of a query expression, and its arguments are *the types of the actual operands* rather than the actual operands themselves. In case of the join operator, this function could be defined as

$$\tau_{join}(rel_1, rel_2, (tuple_1 \times tuple_2 \to \underline{bool})) = \underline{rel}(tuple_1 \circ tuple_2)$$

where $\circ$ denotes an operator for concatenation of lists. Hence the specification above defines simultaneously a polymorphic operator **join** for mapping relations and a type operator τ_{join} for mapping the types of relations.

Type operators have been proposed in [Güti 93] for the construction of complex result types; user defined predicates have only been introduced in this paper. Both techniques essentially provide the power of a general purpose programming language within specifications for type checking or construction of result types, either by predicative or by functional/procedural programming. In this paper only predicates are used for the construction of result types.

As a final specification technique we need *indexed variables*. They are related to list operands and allow one to express that each instance of a type variable in a list is bound independently. This can be illustrated by an operator for (generalized) relational projection:

$\forall\ rel_1$: $\underline{rel}(tuple)$ **in** REL, $\forall\ rel_2$: $\underline{rel}(\underline{tuple}(list))$ **in** REL,
$\forall\ name_i$ **in** $\underline{ident}$, $\forall\ data_i$ **in** DATA, *has2Columns*($list$, $[name_i]$, $[data_i]$):
$rel_1 \times (name_i \times (tuple \to data_i))^+ \quad \to rel_2$ **project** _#[_]

Here the second operand for the **project** operator is a list with one or more elements. Each list element is a pair, consisting of a name (for an attribute of the result relation) and a function computing a value of some data type for a given tuple. The use of indexed variables $name_i$ and $data_i$ means that for each element a different name and a different data type can be chosen. Hence for a list of length n such a specification introduces an array or list of variables, e.g. $name_1, ..., name_n$. We refer to each individual variable as $name_i$ and to the whole list of variables as $[name_i]$. The predicate *has2Columns* makes sure that *list* is a list of pairs such that the projection on all first components is equal to the list $[name_i]$ and the projection on the second components is equal to $[data_i]$. Since $[name_i]$ and $[data_i]$ are bound by an application of **project**, the predicate is in fact used to compute the result type.

2.4 Programs

As mentioned in the introduction, an SOS system accepts programs of the model level and translates them through the use of optimization rules to the representation level. There is a single DDL and DML for programs at both levels. The language consists of five statements:

type	<identifier>	=	<type expression>
create	<identifier>	:	<type expression>
update	<identifier>	:=	<value expression>
delete	<identifier>		
query	<value expression>		

The first statement assigns a name to a type. The type expression is a type of the corresponding type system but may contain names of previously defined named types. In evaluating the statement the respective types (terms) are substituted for the names. The second statement creates a named object of the type denoted by the type expression; its value is undefined. The third statements assigns a value to an object; the value must match the type of the object. The fourth statement deletes an object, the fifth statement returns the value of an expression (a term built from operators) to the user or the calling program.

2.5 Subtype Specification and Database Quantification

Two additional specification techniques are available. A *subtype specification* allows one to introduce subtype polymorphism in addition to the parametric polymorphism achieved by quantification over kinds. As an example, suppose we have a type constructor for arrays, as mentioned above, defined as

$$\mathrm{DATA} \times \underline{integer} \quad \to \mathrm{ARRAY} \quad \underline{array}$$

but want to define "generic" operations that work for arrays of any size. We can introduce a second type constructor for such generic arrays:

$$\mathrm{DATA} \quad \to \mathrm{FIELD} \quad \underline{field}$$

By a subtype specification we can relate pairs of types (type terms), for example:

SUBTYPE $\underline{array}(d, i) < \underline{field}(d)$

The arguments to the type constructors are variables; any variable occurring in the supertype (right hand side) must also occur in the subtype (left hand side) but not vice versa. In practice, this means one can "forget" some type information of the subtype in the supertype. The semantics is that any operation defined for the supertype will be applicable to the subtype as well.

The purpose of *database quantification* is to ensure that certain types in a database can only be defined if certain other types have been defined already (and so to enforce some kind of referential integrity). An *SOS-database* is a pair (*T*, *O*) where *T* is a set of named types and *O* a set of named objects (objects are values of types); these are exactly the types defined and objects created but not deleted by the three commands **type**, **create**, and **delete** described in Section 2.4. A database quantification lets a type variable range not over all possible types in a kind, but only over the named types in the database. This is written as, for example:

$\forall$ *rel* **in extension**(REL)

Hence *rel* must be one of the defined relation types (schemas) in the database.

Database quantification can be viewed as a way to access the database schema (catalog) by restricting the bindings of variables to types that have been defined explicitly. Sometimes it is also necessary to access the catalog through predicates (for implementing type checking predicates in specifications). For this purpose, there exist two system predicates

isType(*name*, *type*)
isObject(*name*, *type*)

which are true, if a type *type* has been defined called *name*, and if an object called *name* of type *type* has been created and not deleted, respectively. For example, after a statement

type person_rel = <u>rel</u>(<u>tuple</u>(<(name, <u>string</u>), (age, <u>integer</u>)>))

has been executed, we have

isType(person_rel, *rel*(*tuple*(<(name, *string*), (age, *integer*)>)))

3 The Type System of GraphDB

In this section we recall the *GraphDB data model*, as described in [Güti 94a, Güti 94b], and "translate" it into an SOS specification of a corresponding *GraphDB type system*. The GraphDB data model distinguishes simple, link, and path classes. Objects of simple classes are similar to objects of other object-oriented models, i.e. they have an identity and a state. The state is described by a tuple. Tuples consist of attributes which are either values of certain (atomic) data types or object identifiers denoting objects. All object identifiers corresponding to objects of a single class are the extension of a reference type corresponding to that class.

Basically the type of an object is described by the type of the class to which the object belongs. For a formal description of the type system it is essential to distinguish the type of a class and the type of an object. We call this an *object type* in contrast to [Güti 94a], where the term object type is used to denote what we call *reference type*.

In contrast to other object-oriented databases, objects of simple classes are also used as nodes in database graphs. The edges of such database graphs are defined by link classes. Objects of link classes are called link objects. These objects have at least two components - one for the source and one for the target object of the edge. In addition the state of link objects is described by a tuple type. Path objects store a list of references to simple and link objects (i.e. nodes and edges) defining a path in the database graph. The state of these objects is again described by a tuple type. Besides the possibilities to describe graph structures and paths by simple, link, and path classes, the GraphDB type system offers inheritance of classes.

3.1 Data Types, Reference Types, and Tuple Types

As mentioned above, a tuple describing the state of an object is an aggregation of values of "atomic" data types and of references to objects. The data types are organized in a hierarchy which is shown in Fig. 2. Besides standard types like STRING, INTEGER, REAL, and BOOL, GraphDB provides the geometric types POINTS, LINES, and REGIONS which are defined in the ROSE algebra [GüSc 93].

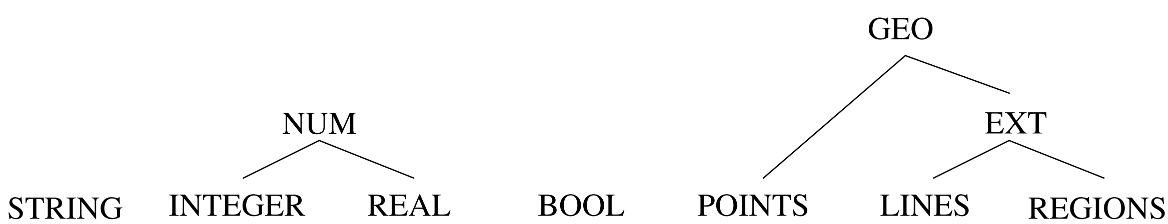

Figure 2: The atomic data types

The purpose of the data type hierarchy is to allow for the definition of polymorphic operations. This can either be done by parametric or by subtype polymorphism. In the ROSE algebra, geometric operations are defined using parametric polymorphism, by introducing GEO and EXT as kinds, and *points*, *lines*, and *regions* as type constructors, in the following way:

$\rightarrow$ EXT	*lines*, *regions*
$\rightarrow$ GEO	*points*, *lines*, *regions*

On the other hand, in the GraphDB data model it is possible to compute for any two related data types (meaning two types belonging to the same tree) a *smallest common supertype*. This is also possible for reference types (if the corresponding object classes have a common ancestor in the class hierarchy) and, by extension, for tuple types. This means EXT and GEO, for example, are not only needed as kinds, but also as types. This leads to the following signature for the atomic data types:

$\rightarrow$ DATA	<u>string</u>, <u>integer</u>, <u>real</u>, <u>num</u>, <u>bool</u>, <u>points</u>, <u>lines</u>, <u>regions</u>, <u>ext</u>, <u>geo</u>
$\rightarrow$ NUM	<u>integer</u>, <u>real</u>, <u>num</u>
$\rightarrow$ EXT	<u>lines</u>, <u>regions</u>, <u>ext</u>
$\rightarrow$ GEO	<u>points</u>, <u>lines</u>, <u>regions</u>, <u>ext</u>, <u>geo</u>

Reference types correspond in GraphDB directly to class names, i.e. each reference type denotes objects of a single class. The extension of a reference type is the set of all object identifiers denoting objects of this class. For each class which is defined for a database there is a corresponding reference type. We model reference types by terms of a kind REF. We can define such terms by a type constructor <u>oid</u> taking an identifier as operand:

<u>ident</u> $\rightarrow$ REF <u>oid</u>

For example, we denote the reference type corresponding to class *departure* by <u>oid</u>(departure).

As usual, *tuple types* are described by a list of attribute names and corresponding types. These types can be data types and reference types. In GraphDB the usual subtype relationships are defined for tuple types [Güti 94a].

Tuple types are described by terms of kind TUPLE. These terms are generated by the <u>tuple</u> type constructor. This type constructor takes a list of pairs consisting of an identifier and a reference type or an atomic data type. There may be no duplicate attribute names in a single tuple type:

$\forall$ *list* **in** (<u>ident</u> $\times$ (REF $\cup$ DATA))*, *noDuplicateNames*(*list*):
list $\rightarrow$ TUPLE <u>tuple</u>

3.2 Object Types and Classes

As mentioned above, there are simple, link, and path classes. All these kinds of classes can be arranged in individual simple, link, and path class hierarchies. However, different kinds of classes are not related via inheritance. Multiple inheritance is not supported.

In our specification of the type system of GraphDB we distinguish object types and class types. Similar to a tuple type describing the individual tuples of a relation, object types describe the type of individual objects. However, the instance of a class type is the collection of all objects of the object type corresponding to the class. Hence, in the formal specification of the type system of GraphDB there are simple classes, link classes, and path classes and the corresponding simple, link, and path object types. We model this by individual kinds and corresponding type constructors for the different object and class types.

3.2.1 Simple Object Types and Classes

Simple classes are defined by a tuple type and a class name. Using the GraphDB DDL we may for example define simple classes *vertex* and *station* as follows:

```
class vertex = pos: POINT[1]
class station = name: STRING, loc: vertex
```

Each simple class can be used as base class to derive further classes. In derived classes the usual modifications of the tuple type are allowed, i.e. attribute types of the tuple type corresponding to the base class can be replaced by subtypes and new attributes can be added to the tuple type. For example, the definition

```
vertex class city = name: STRING, pop: INTEGER
```

defines *city* objects with an inherited *pos* attribute and two additional attributes.

We define simple object types as types of a kind SOBJECT. Terms of kind SOBJECT are constructed by the *object* type constructor. This type constructor takes the reference type corresponding to the class to which the object belongs and a tuple type describing the possible states of the object as operands:

$$\text{REF} \times \text{TUPLE} \quad \rightarrow \text{SOBJECT} \quad \underline{\textit{object}}$$

The *object* type constructor corresponds to the intuitive view of an object as having an identity (an element of the carrier set of a reference type is an object identifier) and a state (a tuple of atomic values and object identifiers).

A simple class type is a type of kind SCLASS. We describe these types by the *class* type constructor, which takes the object type of the class and the name of its superclass as operands:

$$\forall\ obj\text{: } \underline{object}(\underline{oid}(name), \_)\ \textbf{in}\ \text{SOBJECT:}$$
$$name \times obj \quad \rightarrow \text{SCLASS} \quad \underline{class}$$

$$\forall\ \_\text{: } \underline{class}(\_, \underline{object}(\underline{oid}(name), tuple_1))\ \textbf{in extension}(\text{SCLASS}),$$
$$\forall\ obj\text{: } \underline{object}(\_, tuple_2)\ \textbf{in}\ \text{SOBJECT},\ subtype\ (tuple_2, tuple_1)\text{:}$$
$$name \times obj \quad \rightarrow \text{SCLASS} \quad \underline{class}$$

There are two different kinds of simple class types. The first kind of classes has no superclass, i.e. the class is the root of an inheritance hierarchy. Formally, the name of the superclass is the name of the class itself. This case is covered by the first specification of the *class* type constructor. (The underscore denotes an anonymous variable).

The second kind of classes inherits from other classes. This case is covered by the second specification of the *class* type constructor. Here the variable definition

$$\forall\ \_\text{: } \underline{class}(\_, \underline{object}(\underline{oid}(name), tuple_1))\ \textbf{in extension}(\text{SCLASS})$$

only allows bindings for the variables whenever there is a simple class in the database having object type "$\underline{object}(\underline{oid}(name), tuple_1)$", i.e. where a class with name "*name*" has been defined. Furthermore, we ensure that a subclass has a tuple type which is a subtype of the tuple type of the superclass. This is done by the predicate *subtype*.

Example: The classes *vertex* and *city* are described by the types

class (vertex, *object*(*oid*(vertex), *tuple*(<(pos, *points*)>)))
class (vertex, *object*(*oid*(city), *tuple*(<(pos, *points*), (name, *string*), (pop, *integer*)>)))

Our algebraic description of objects and classes requires that all attributes are listed in the object

1. We write POINT and LINE in the GraphDB DDL to indicate that the application needs a single point or a simple polyline as an attribute value. The actual data types from the ROSE algebra are capable of representing sets of points and line segment graphs, but of course can also represent these simple values.

type of the class. There is no notion for inherited attributes. The disadvantage of this approach is that the description of objects and classes may be lengthy. The advantage is that we need not consider the effects of inheritance on attributes during the specification of operators.

3.2.2 Link Classes

As mentioned above, link classes describe edges of database graphs. Hence, the definition of a link class has to specify the type of the source and the target object. The following DDL statement describes a link class *arc* connecting two *vertex* objects:

```
link class arc = route: LINE from vertex to vertex
```

Objects of class *arc* may be used to describe arcs in a highway or a railway network. The attribute *route* describes the shape of *arc* objects by a polyline. It is obvious that the source and the target class which represent vertices in the database graph must be simple classes. Again we can derive further classes from a defined link class. In these subclasses the tuple type may be modified in the usual way. But we may also replace the source or the target class by subclasses.

In the GraphDB algebra we again describe the type of individual link objects and the type of link classes. Types of kind LOBJECT which denote link object types are defined by the type constructor <u>*linkObject*</u> which takes the following operands:

- the reference type of the link class corresponding to the link object type
- the reference type of the class containing the source objects
- the reference type of the class containing the target objects
- the tuple type describing the state of an object

$\forall$ _: <u>*class*</u>(_, <u>*object*</u>(ref_1, _)) **in extension**(SCLASS),
$\forall$ _: <u>*class*</u>(_, <u>*object*</u>(ref_2, _)) **in extension**(SCLASS),
REF $\times$ ref_1 $\times$ ref_2 $\times$ TUPLE $\rightarrow$ LOBJECT <u>*linkObject*</u>

In the specification we ensure that the reference types defining the source and the target objects denote classes which have been defined previously.

The elements of the carrier sets of types of kind LOBJECT are quadruples (o_1, o_2, o_3, t). o_1 is the object identifier of the link object, o_2 is the object identifier of the source object, o_3 is the object identifier of the target object, and t is a tuple of atomic values and object identifiers.

A link class contains all objects of the link object type corresponding to this class. Link classes are types of kind LCLASS. We construct theses types by the <u>*linkClass*</u> type constructor. This type constructor takes the object type of the class and the name of its superclass as operands:

$\forall$ *obj*: <u>*linkObject*</u>(<u>*oid*</u>(*name*), _, _, _) **in** LOBJECT:
name $\times$ *obj* $\rightarrow$ LCLASS <u>*linkClass*</u>

$\forall$ _: <u>*linkClass*</u>(_, <u>*object*</u>(<u>*oid*</u>(*name*), $source_1$, $target_1$, $tuple_1$)) **in extension**(LCLASS),
$\forall$ *obj*: <u>*linkObject*</u>(_, $source_2$, $target_2$, $tuple_2$) **in** LOBJECT,
subtype($tuple_2$, $tuple_1$), *subclass*($source_2$, $source_1$), *subclass*($target_2$, $target_1$):
name $\times$ *obj* $\rightarrow$ LCLASS <u>*linkClass*</u>

The first specification of the <u>*linkClass*</u> type constructor describes link classes at the root of a link class hierarchy. The second specification considers inheritance of link classes. If we derive a link class from an existing link class, this class must already exist. This is ensured by a corresponding variable definition. The predicates *subclass*($source_2$, $source_1$) and *subclass*($target_2$, $target_1$) test that the new source and the new target class are subclasses of the source and the target class of the link class from which the new class inherits. Note that the information needed in the *subclass* predicate can be obtained from the "system catalog" via the predicate *isType*, for example by one of the rules for *subclass*:

subclass(*oid*(*sub*), *oid*(*super*)) :- *isType*(_, *class*(*super*, *object*(*oid*(*sub*), _))).

Example: The class *arc* is described by

linkClass (arc, *linkObject*(*oid*(arc), *oid*(vertex), *oid*(vertex), *tuple*(<(route, *lines*)>)))

3.2.3 Path Classes

Simple classes and link classes define a so-called *database schema graph*. This graph has a node for each simple class of the database schema and an edge for each link class of the database schema. Similarly, there exists a *database instance graph*; its nodes are the objects of simple classes and its edges the objects of link classes. Path classes describe objects with an associated path in the database instance graph (e.g. highways are such objects). In addition to the class name and the tuple type, path objects are defined by *path types* describing the possible structures of paths in the database instance graph. A path type is basically a finite automaton belonging to a regular expression over link class names. In Fig. 3 we show the path type corresponding to the regular expression arc^+. A circle around a node indicates the start node, a square one of the final nodes.

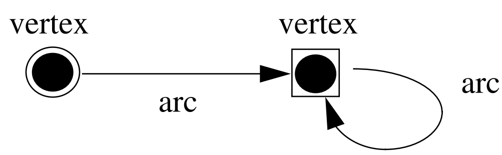

Figure 3: A sample path type

In GraphDB a path class based on the path type of Fig. 3 can be defined as follows:

```
path class phys_route as arc+
```

One can derive subclasses of a path class by modifying the tuple type.

Again in the GraphDB algebra we distinguish path objects and path classes. Path objects are described by path object types which are types of kind POBJECT. These object types are generated by the *pathObject* type constructor. This type constructor takes the reference type corresponding to the class to which the object belongs and a tuple type as operands. In addition, *pathObject* requires a path type - a type of a kind PATH - describing the finite automaton which must be compatible with the database schema graph.

We first consider path types. The reference types corresponding to a link class and its source and target classes can be used to obtain a path type via a new *link* type constructor. Since path types are equivalent to regular expressions over link class names, further path types are constructed by the type constructors *plus*, *concat*, *or*, and *star*. These constructors correspond to the usual operators for regular expressions:

$\forall$ _: *linkClass*(_, *linkObject*(*ref*, *source*, *target*, _)) **in extension**(LCLASS):
$ref \times source \times target \rightarrow$ PATH *link*

$\forall$ $path_1$ **in** PATH, $\forall$ $path_2$ **in** PATH, $\forall$ *ref* **in** REF, *to*($path_1$, *ref*), *from*($path_2$, *ref*):
$path_1 \times path_2 \rightarrow$ PATH *concat*

$\forall$ *path*$_1$ **in** PATH, $\forall$ *path*$_2$ **in** PATH, $\forall$ *ref* **in** REF, *from*(*path*$_1$, *ref*), *from*(*path*$_2$, *ref*):
path$_1 \times$ *path*$_2$ $\rightarrow$ PATH *or*

$\forall$ *path* **in** PATH, $\forall$ *ref* **in** REF, *from*(*path*, *ref*), *to*(*path*, *ref*):
path $\rightarrow$ PATH *star*, *plus*

Since path types describe finite automatons there is one source node and a set of target nodes for each path type. *concat* connects two path types where the set of target nodes of the first path type contains one reference type and the source node of the second path type is this reference type. *or* combines path types starting with the same reference type. *star* and *plus* can be applied to each path type having one target node which is identical to the source node of the path type. For these specifications predicates *from* and *to* are required. Predicate *from* tests whether a given reference type *ref* is the source node of the automaton described by a given path type *path*. Predicate *to* tests whether the finite automaton described by a given path type *path* has a single target node and whether this node is denoted by reference type *ref*. Since the predicates *to* and *from* can only be defined if the path type contains the reference type of the source and target nodes of each edge of the path, the basic type constructor *link* takes three reference types as operands.

Using types of kind PATH we can easily specify a type constructor *pathObject* for path objects:

REF $\times$ PATH $\times$ TUPLE $\rightarrow$ POBJECT *pathObject*

An element of a carrier set corresponding to a type of kind POBJECT consists of an object identifier, a tuple, and a path in the database graph which is given by a sequence of identifiers of simple and link objects.

Path classes are types of kind PCLASS. We construct these types by the *pathClass* type constructor. This type constructor takes the object type of the path class and the name of its superclass as operands:

$\forall$ *obj*: *pathObject*(*oid*(*name*), _, _) **in** POBJECT:
name $\times$ *obj* $\rightarrow$ PCLASS *pathClass*

$\forall$ _: *pathClass*(_, *pathObject*(*oid*(*name*), *path*, *tuple*$_1$)) **in extension**(PCLASS),
$\forall$ *obj*: *pathObject*(_, *path*, *tuple*$_2$) **in** POBJECT, *subtype*(*tuple*$_2$, *tuple*$_1$):
name $\times$ *obj* $\rightarrow$ PCLASS *pathClass*

The first part of the specification considers path classes at the top of a path class inheritance hierarchy. In the second part we consider the inheritance of path classes. In this specification we again ensure that the superclass exists. Furthermore we test the subtype relationship of the tuple types *tuple*$_2$ and *tuple*$_1$. The path type of the subclass is the path type of its superclass.

Example: The path class *phys_route* is described by the type

pathClass(phys_route, *pathObject*(*oid*(phys_route),
plus(*link*(*oid*(arc), *oid*(vertex), *oid*(vertex))), *tuple*(<>)))

3.2.4 Abstract Objects and Classes

To facilitate the specification of the operators we introduce two additional kinds OBJECT and CLASS which subsume simple, link, and path objects and simple, link, and path classes. This abstraction reduces these objects and classes to their basic constituents.

Object types of kind OBJECT are generated by the *abstractObject* type constructor which takes the reference type of the class to which the object belongs and a tuple type describing the possible states of the object as operands:

$$\text{REF} \times \text{TUPLE} \quad \rightarrow \text{OBJECT} \quad \underline{abstractObject}$$

Types of kind CLASS are generated by the type constructor <u>*abstractClass*</u>. This type constructor takes the object type of the class and the name of its superclass as operands. Types of kind CLASS are abstract classes, i.e. there are no database objects of such a type. Hence, we need only specify the kinds of the operands of <u>*abstractClass*</u>:

$$\underline{ident} \times \text{OBJECT} \quad \rightarrow \text{CLASS} \quad \underline{abstractClass}$$

The different kinds of objects types and classes are related by a subtype specification:

SUBTYPE

<u>*object*</u> (*ref*, *tuple*) < <u>*abstractObject*</u> (*ref*, *tuple*)
<u>*linkObject*</u> (*ref*, _, _, *tuple*) < <u>*abstractObject*</u> (*ref*, *tuple*)
<u>*pathObject*</u> (*ref*, _, *tuple*) < <u>*abstractObject*</u> (*ref*, *tuple*)

<u>*class*</u> (*name*, *obj*) < <u>*abstractClass*</u> (*name*, *obj*)
<u>*linkClass*</u> (*name*, *obj*) < <u>*abstractClass*</u> (*name*, *obj*)
<u>*pathClass*</u> (*name*, *obj*) < <u>*abstractClass*</u> (*name*, *obj*)

3.3 Example: The Public Transport Network

As an example for a GraphDB database we briefly review the definition of the public transport network given in [Güti 94a]. This network consists of three layers: the physical network, the lines layer, and the time schedules layer. The physical layer is modeled by the classes *vertex*, *arc*, and *phys_route* which we have already discussed in Section 3.2:

```
class vertex = pos: POINT;
link class arc = route: LINE from vertex to vertex;
path class phys_route as arc+;
```

The *lines layer* is used to describe regular connections over the physical network such as bus or underground lines. Lines are modelled by a link class *connection* describing the characteristics of a connection of two *stations*. A *line* is a sequence of connections, i.e. a path class with the path type *connection*$^+$. Note that a connection has a reference to a corresponding path of the underlying physical network.

```
class station = name: STRING, loc: vertex;
link class connection = travel_minutes: INTEGER, way: phys_route
     from station to station;
path class line = line_type: STRING, line_no: INTEGER as connection+;
```

The third layer describes the time schedule as a graph over *departure* and *arrival events* which are subclasses of a class *event*. Link objects of class *travel* lead from a departure event to an arrival event; they describe traveling on a single “carrier” e.g. a train. *Stay* edges model a train between arriving at a station and its next departure. *Change* and *wait* edges allow one to switch at a station; the *change* edge connects an arrival with the next departure of any train at this station. *Wait* edges connect departures in the order of departure time. Trips of specific trains are stored in the database as path objects of type *travel (stay travel)** (note that such a stored path includes the nodes, i.e. the departure and arrival events). Changing the line at a station is described by a path of type *change wait**.

```
class event = time: INTEGER, at_station: station, of_line: line;
event class arrival, departure;
link class travel = through: connection from departure to arrival;
link class stay from arrival to departure;
link class change from arrival to departure;
link class wait from departure to departure;
path class trip as travel (stay travel)*;
```

We can now define the classes of the public transport network using the SOS-DDL and the GraphDB type system as introduced above. For each class one needs to define an object type *o* and create a class with a class type corresponding to *o*. For brevity, we only show a few definitions for the third layer.

type event_type = *object*(*oid*(event),
tuple(<(time, *integer*), (at_station, *oid*(station)), (of_line, *oid*(line))>))
create event: *class*(event, event_type)

type departure_type = *object*(*oid*(departure),
tuple(<(time, *integer*), (at_station, *oid*(station)), (of_line, *oid*(line))>))
create departure: *class*(event, departure_type)

type travel_type = *linkObject*(*oid*(travel), *oid*(departure), *oid*(arrival),
tuple(<(through, *oid*(connection))>))
create travel: *linkClass*(travel, travel_type)

type trip_path = *concat*(*link*(*oid*(travel), *oid*(departure), *oid*(arrival)),
star(*concat*(*link*(*oid*(stay), *oid*(arrival), *oid*(departure)),
link(*oid*(travel), *oid*(departure), *oid*(arrival)))))
type trip_type = *pathObject*(*oid*(trip), trip_path, *tuple*(<>))
create trip: *pathClass*(trip, trip_type)

4 Algebra Operators for Querying

In this section we develop an algebra over the type system of the previous section to model the querying facilities of the GraphDB data model.

4.1 Basic Features of the Query Language

Let us start with a brief overview of querying in GraphDB. A query generally consists of several steps, $Q = q_1; \ldots ; q_m$. In each step q_i one or more classes of simple or link objects can be computed. These classes are added to the database graph after step q_i, and step q_{i+1} considers the modified database graph. Hence it is possible to change the database graph within a query. There are four kinds of conceptual structures that can be manipulated in queries:

- homogeneous sequences of objects
- heterogeneous sequences of objects
- (single) objects
- values of (atomic) data types

The most simple way to obtain a *homogeneous sequence* is by writing a class name. This creates a sequence of all objects belonging to the class, in some unspecified order. A *heterogeneous sequence* can be obtained by writing several class names in angular brackets. The resulting sequence contains all objects of the respective classes. Again the order is not specified. Such a sequence can be viewed as describing a graph. A path associated with a path object, or computed in a query, can also be considered as a heterogeneous sequence of objects. In this case the order is defined and matches some path type. The basic tools for querying a database are:

- The *derive* statement, which takes the role of the classical *select … from … where.*
- The *rewrite operation* which supports the manipulation of heterogeneous sequences.
- The *union operation* which transforms a heterogeneous sequence of objects into a homogeneous sequence having a tuple type which is a common supertype.
- A collection of *graph operations*.

4.2 Derive

The *derive* statement corresponds to the well-known *select … from … where* but extends it in that one can refer to connections in the database graph. It also allows one to construct new objects, in particular, new link objects, and so to extend the database graph.

4.2.1 An Example

Let us consider an example based on the public transport database of Section 3.3.

> Q1. Make a listing of all departures from Dortmund main station, showing time of departure, type and number of train, and end station and arrival time there.

```
on departure at_station station, departure of_line line,
     departure in trip
where station.name = "Dortmund"
derive dtime: departure.time, line.line_type, line.line_no,
     (trip end).at_station.name, atime: (trip end).time
```

Here in the on-clause all combinations of *departure*, *station*, *line*, and *trip* objects are formed where *station* is the *at_station* attribute value of the *departure* object, *line* is the value of its *of_line* attribute, and *departure* is a node in the path of the *trip* object. Conceptually, the on-clause generates a *sequence of 4-tuples of objects* fulfilling these conditions. This sequence is filtered in the where-clause; only quadruples whose *station* component object fulfills the where-condition pass this filter. Finally, the derive-clause constructs new objects, one object for each quadruple it receives from the previous steps. In this case, unnamed simple objects with five attributes are constructed; the attribute value and type are described by an expression, and the attribute name is either defined (e.g. *dtime*) or inherited from an attribute name of one of the objects of the quadruple. It is also possible to create named objects and link objects, or to return one of the objects received by the derive-clause. For a complete description see [Güti 94a]. In any case, the output from a derive statement as a whole is a *homogeneous sequence of objects* which can be manipulated by further operations.

In the following sections we introduce a type for homogeneous sequences and operators to create them (realizing the on-clause), define operators for accessing objects and attribute values (needed in the where- and derive-clauses), define selection to implement the where-clause, and introduce operators to create simple objects and link objects to realize the derive-clause.

4.2.2 Generating Homogeneous Sequences

As mentioned in Section 4.1, the user manipulates in queries *homogeneous sequences of objects* (as well as the other three conceptual structures). However, within the scope of the derive-statement, *homogeneous sequences of tuples of objects* are needed. In the algebra we therefore define this more general type and treat homogeneous sequences of objects as a special case (where a tuple of objects contains just one object).

Homogeneous sequences are described by types of a kind SEQ. The type constructor <u>*seq*</u> takes a list of pairs of unique identifiers and arbitrary object types as operands:

$\forall$ *list* **in** (<u>*ident*</u> $\times$ OBJECT)*, *noDuplicateNames*(*list*):

list $\rightarrow$ SEQ <u>*seq*</u>

The most simple way to generate a homogeneous sequence of objects in a query is by writing a class name. The following specification defines each class name as a corresponding operator.

$\forall$ *obj*: <u>*abstractObject*</u>(<u>*oid*</u>(*name*),_) **in extension**(OBJECT),
$\forall$ _: <u>*abstractClass*</u>(_, *obj*) **in extension**(CLASS),
$\forall$ *seq*: <u>*seq*</u>(<(*name*, *obj*)>) **in** SEQ:

$\rightarrow$ *seq* *name*

The **on**-clause of a query contains a *subgraph specification* given as a list of *patterns*. Each pattern introduces a single variable or several related variables denoting objects. There are four possible relationships:

(a) An object o_1 is referenced by an attribute of another object o_2.
(b) A link object l connects two simple objects o_1 and o_2.
(c) A simple object s is contained in a path p.
(d) A link object l is contained in a path p.

As we have seen above, the **on**-clause constructs a sequence of tuples of objects. The number of components of these tuples is the number of variables in the **on**-clause. Each tuple contains objects satisfying all relations of the **on**-clause. The variables in the **on**-clause are class names and range over the set of objects corresponding to that class. It is also possible to introduce explicit variables, for example

```
on station(s1) connection station(s2)
```

where *s1* and *s2* are new variables denoting station objects. Here the **on**-clause generates all triples of two *station* objects *s1, s2*, and a *connection* object *c* where *c* connects *s1* and *s2* (relationship (b)).

The **on**-clause and its subgraph specification are modeled by an operator **on** which takes all classes describing a subgraph and a formalized subgraph specification as operands. Patterns of a subgraph specification are described by algebra expressions whose result types are of a kind PAT. There are three type constructors for types of kind PAT and three corresponding operators describing the four possible relationships described above.

Subgraph specifications of type (a) are described by expressions whose result type is constructed by the <u>*attrSpec*</u> type constructor. This type constructor has three operands: a reference type, an attribute name of the tuple type of the class corresponding to the first reference type, and the reference type of the attribute whose name is given as the second operand:

REF $\times$ <u>*ident*</u> $\times$ REF $\rightarrow$ PAT <u>*attrSpec*</u>

Expressions describing the first type of subgraph specification basically consist of an application of the **aspec** operator. This operator takes three identifiers and returns a subgraph specification whose type is described by an application of the <u>*attrSpec*</u> type constructor:

$\forall$ *seq* **in** SEQ, $\forall$ obj_1: <u>abstractObject</u>(ref_1, *tuple*) **in** OBJECT,
$\forall$ obj_2: <u>abstractObject</u>(ref_2, _) **in** OBJECT,
$\forall$ $name_1$ **in** <u>ident</u>, $\forall$ $name_2$ **in** <u>ident</u>, $\forall$ *spec*: <u>attrSpec</u>(ref_1, $name_3$, ref_2) **in** PAT,
member($name_1$, obj_1, *seq*), *member*($name_2$, obj_2, *seq*), *member*($name_3$, ref_2, *tuple*):
$seq \times name_1 \times name_3 \times name_2 \quad \rightarrow spec$ **aspec** _#(_,_,_)

The name associated with a class in a sequence may change due to applications of the **rename** operator which is specified below. Hence, it is not sufficient to define the **aspec** operator and the remaining operators for subgraph specifications with respect to the class names stored in the database but we have to consider a sequence operand. The specification of the **on** operator given below ensures that the right sequence is passed to the **aspec** operator.

Example: The relationship “departure at_station station” on an operand sequence *sequence* is expressed by a term

sequence **aspec**(departure, at_station, station)

of the algebra which is of type <u>attrSpec</u>(<u>oid</u>(departure), at_station, <u>oid</u>(station)).

To describe the second type of subgraph specifications we define the <u>linkSpec</u> type constructor and the corresponding **lspec** operator. The type constructor takes three reference types as operands. The first reference type denotes a link object type, and the other two denote the simple object types corresponding to the source and the target object types of the link object type:

$\forall$ *link*: <u>linkObject</u>(ref_1, ref_2, ref_3, _) **in** LOBJECT:
$ref_2 \times ref_1 \times ref_3 \quad \rightarrow$ PAT <u>linkSpec</u>

Similar to the **aspec** operator the operands of the **lspec** operator are three identifiers. The result of an application of **lspec** is a subgraph specification whose type is described by an application of the <u>linkSpec</u> type constructor.

$\forall$ *seq* **in** SEQ, $\forall$ obj_1: <u>linkObject</u>(ref_1, ref_2, ref_3, _) **in** LOBJECT,
$\forall$ obj_2: <u>object</u>(ref_2, _) **in** SOBJECT, $\forall$ obj_3: <u>object</u>(ref_3, _) **in** SOBJECT,
$\forall$ $name_1$ **in** <u>ident</u>, $\forall$ $name_2$ **in** <u>ident</u>, $\forall$ $name_3$ **in** <u>ident</u>,
$\forall$ *spec*: <u>linkSpec</u>(ref_2, ref_1, ref_3) **in** PAT,
member($name_1$, obj_1, *seq*), *member*($name_2$, obj_2, *seq*), *member*($name_3$, obj_3, *seq*):
$seq \times name_2 \times name_1 \times name_3 \quad \rightarrow spec$ **lspec** _ #(_,_,_)

The subgraph specification which tests whether a link or a simple object is contained in a path object is described by expressions whose result type is constructed by the type constructor <u>pathSpec</u>. The corresponding operator is called **pspec**. <u>pathSpec</u> takes two operands. One operand is a reference type denoting the simple or link object type and the second operand is a path type:

$\forall$ *ref* **in** REF, $\forall$ _: <u>pathObject</u>(_, *path*,_) **in** POBJECT:
$ref \times path \quad \rightarrow$ PAT <u>pathSpec</u>

The **pspec** operator takes two identifiers as operands. The first identifier describes the simple or the link object type which must be contained in the path type denoted by the second operand:

$\forall$ *seq* **in** SEQ,
$\forall$ obj_1: <u>abstractObject</u>(*ref*, _) **in** (SOBJECT $\cup$ LOBJECT),
$\forall$ obj_2: <u>pathObject</u>(_, *path*, _) **in** POBJECT,
$\forall$ $name_1$ **in** <u>ident</u>, $\forall$ $name_2$ **in** <u>ident</u>, $\forall$ *spec*: <u>pathSpec</u>(*ref*, *path*) **in** PAT,
member($name_1$, obj_1, *seq*), *member*($name_2$, obj_2, *seq*), *inPath*(*ref*, *path*):
$seq \times name_1 \times name_2 \quad \rightarrow spec$ **pspec** _ #(_,_)

The predicate *inPath* ensures that the reference type *ref* denotes a simple or a link object type which is part of the path described by the path type *path*.

Finally, we specify the **on** operator:

$\forall$ $list_1$ **in** SEQ^+, $\forall$ pat_i **in** PAT, $\forall$ *seq*: <u>*seq*</u>($list_2$) **in** SEQ,
concat($list_1$, $list_2$):

$list_1 \times (seq \rightarrow pat_i)^*$ $\rightarrow seq$ **on** _#[_]

The first operand is a list of homogeneous sequences (each can be generated by writing a class name), the second a list of mappings; each mapping yields a pattern for a given sequence. The patterns together describe a subgraph specification. The predicate *concat* ensures that $list_2$ denotes the concatenation of all lists describing sequence types within $list_1$. Recall that $list_1$ is a list of sequence types of the form <u>*seq*</u>(<(*name*, *obj*)>), hence the "tuple of object" types of all sequences have to be concatenated to form the type of $list_2$. The indexed variables pat_i express that each of the mappings has its own result type. The on-clause from the example of Section 4.2.1

```
on departure at_station station, departure of_line line,
     departure in trip
```

can now be expressed as:

type seq = <u>*seq*</u>(<(departure, departure_type), (station, station_type), (line, line_type),
(trip, trip_type)>)
query <departure, station, line, trip>
on[<**fun**(s: seq) s **aspec**(departure, at_station, station),
fun(s: seq) s **aspec**(departure, of_line, line),
fun(s: seq) s **pspec**(departure, trip)>]

In this example, the first operand of the **on** operator has type

<<u>*seq*</u>(<(departure, departure_type)>), <u>*seq*</u>(<(station, station_type)>),
<u>*seq*</u>(<(line, line_type)>), <u>*seq*</u>(<(trip, trip_type)>)>

whereas the result sequence of the **on** operator has type *seq* as in the type definition statement above. In fact, because an SOS parser can infer the type *seq* from the definition of the **on** operator (as for **join** in Section 2.3), one can omit the type definition and formulate the query as:

query <departure, station, line, trip>
on[<**fun**(s) s **aspec**(departure, at_station, station),
fun(s) s **aspec**(departure, of_line, line),
fun(s) s **pspec**(departure, trip)>]

In the sequel we shall usually omit explicit type definitions and use the short form whenever possible.

To express the introduction of explicit variables in the algebra, as in the example

```
on station(s1) connection station(s2)
```

we need an operator for renaming. It takes a sequence consisting of objects of a single class and changes the name which is associated with these objects in the sequence:

$\forall$ seq_1: <u>*seq*</u>(<(_, *obj*)>) **in** SEQ, $\forall$ seq_2: <u>*seq*</u>(<(*name*, *obj*)>) **in** SEQ:

$seq_1 \times name$ $\rightarrow seq_2$ **rename** #(_,_) The **on**-clause shown above would then be represented in the algebra as

query <**rename**(station, s1), **rename**(station, s2)>
on[<**fun**(s) s **lspec**(s1, connection, s2)>]

where the inferred type of *s* is

$\underline{seq}$(<(s1, station_type), (s2, station_type)>)

For evaluating an application of the **on**-operator it is sufficient if we let the the carrier set of any type *t* of kind PAT just consist of the terms of that type (each term represents itself).

4.2.3 Access to Individual Objects and Atomic Values

The GraphDB-DML provides access to individual objects of the database by writing a list of attribute values and a class name to identify an object of the class. For example, we may access the station object corresponding to Dortmund by writing

```
station("Dortmund")
```

In the GraphDB-algebra we provide an operator **object** which transforms a sequence of objects into a single object. This is done by evaluating a predicate on all objects of the sequence, which must evaluate to true for exactly one object; otherwise a runtime error occurs. For example, we translate the expression above into:

query station **object**[**fun**(t) t name = “Dortmund”]

The **object** operator is specified as follows:

∀ *seq*: $\underline{seq}$(<_, *obj*>) **in** SEQ:
seq × (*obj* → $\underline{bool}$) → *obj* **object** _#[_]

Hence in the example the type of *t* is inferred to be *station_type*.

The access to the attribute *name* of the station objects is also an operator application. This operator either returns an instance of an object type or an instance of an atomic data type. Operand is the object whose component is accessed. The operator name denotes the attribute:

∀ obj_1: $\underline{abstractObject}$(_, *tuple*) **in** OBJECT, ∀ *name* **in** $\underline{ident}$,
∀ obj_2: $\underline{abstractObject}$(*ref*, _) **in extension**(OBJECT), *member*(*name*, *ref*, *tuple*):
obj_1 → obj_2 *name* _#

∀ *obj*: $\underline{abstractObject}$(_, *tuple*) **in** OBJECT, ∀ *name* **in** $\underline{ident}$,
∀ *dtype* **in** DATA, *member*(*name*, *dtype*, *tuple*):
obj → *dtype* *name* _#

In addition to these two applications of this access by name, we may also use such operators to access an object which is contained in a tuple of objects. This is useful, for example, to evaluate selections on homogeneous sequences of tuples of objects:

∀ *list* **in** ($\underline{ident}$ × OBJECT)*, ∀ *name* **in** $\underline{ident}$, ∀ *obj* **in** OBJECT, *member* (*name*, *obj*, *list*):
list → *obj* *name* _#

4.2.4 The Where-Clause

The next part of the **derive**-statement is the **where**-clause. In the GraphDB-algebra we realize this step by a **select** operator. This operator takes as operands the homogeneous sequence of tuples of objects generated by the **on** operator and a mapping from the tuple of object types defining the sequence to <u>*bool*</u>:

$\forall$ *seq*: <u>*seq*</u>(*list*) **in** SEQ:

$seq \times (list \to \underline{bool})$	$\to seq$	**select**	_#[_]

As an example we consider the query

```
on station where name = "Dortmund" derive station
```

We can now translate the query into the following algebra expression:

query station **select**[**fun**(t) t station name = "Dortmund"]

where *t* is of type <(station, station_type)>.

4.2.5 The Derive-Clause

The final part of a **derive** statement is the **derive**-clause. It has three basic forms [Güti 94a]:

(1) A variable which has been introduced previously is taken as the query result. In this case the **derive** statement is a complex selection.
(2) The result is a new simple class constructed from a list of attribute specifications.
(3) The result is a new link class constructed from two variables denoting simple objects and a list of attribute specifications.

An attribute specification is either an attribute access or a new name and an expression. The attribute access has the form *variable.attribute-name* in which case the new attribute name and the value is taken from the object denoted by *variable*. In the second case the attribute specification has the form *new-name: expression*. In this case the attribute name is *new-name* and the value is obtained by evaluating *expression*. Case (1) is described in the GraphDB-algebra by a **selectClass** operator, case (2) by a **class** operator, and case (3) by a **linkClass** operator.

The **selectClass** operator takes a homogeneous sequence of tuples of objects and transforms it into a homogeneous sequence of objects of just one class. The name given as a second operand must occur in the operand sequence. This is captured by the specification:

$\forall$ seq_1: <u>*seq*</u>(*list*) **in** SEQ, $\forall$ seq_2: <u>*seq*</u>(<(*name*, *obj*)>) **in** SEQ, *member*(*name*, *obj*, *list*):

$seq_1 \times name$	$\to seq_2$	**selectClass**	_#[_]

For each of the three operators **selectClass**, **class**, and **linkClass** there is a second version applicable not to a sequence but to an individual tuple of objects and returning just a single object. These are needed in connection with the **rewrite** operator (Section 4.3). For **selectClass** the corresponding specification is:

$\forall$ *list* **in** ($\underline{ident} \times$ OBJECT)$^+$, $\forall$ *name* **in** <u>*ident*</u>, $\forall$ *obj* **in** OBJECT, *member*(*name*, *obj*, *list*):

$list \times name$	$\to obj$	**selectClass**	_#[_]

The operator **class** takes a homogeneous sequence of tuples of objects and transforms it into a sequence of objects of a simple class. Operands of **class** are the homogeneous sequence, a new class name, and a list of attribute names and mappings. For each mapping, the domain is the list of names and object types corresponding to the homogeneous sequence. The range is a reference type or an atomic type. The tuple type of the resulting object type has an attribute for each mapping in the list of mappings. The attribute type is the range of the corresponding mapping and the attribute name is the corresponding name in the list:

$$\begin{array}{l}
\forall\ seq_1\text{: }\underline{seq}(list_1)\ \textbf{in}\ \text{SEQ},\ \forall\ name_i\ \textbf{in}\ \underline{ident},\ \forall\ refdata_i\ \textbf{in}\ (\text{REF} \cup \text{DATA}),\\
\forall\ seq_2\text{: }\underline{seq}(<(name,\ \underline{object}(\underline{oid}(name),\ \underline{tuple}(list_2)))>)\ \textbf{in}\ \text{SEQ},\\
has2Columns(list_2,\ [name_i],\ [refdata_i])\text{:}\\
\quad seq_1 \times name \times (name_i \times (list_1 \rightarrow refdata_i))^+ \qquad \rightarrow seq_2 \quad \textbf{class} \qquad _\#[_,_]
\end{array}$$

The predicate *has2Columns* has already been explained with the projection operator in Section 2.3; it constructs $list_2$ from the list of names $[name_i]$ and the list of result types $[refdata_i]$. The corresponding version of the class operator for treating single objects is:

$$\begin{array}{l}
\forall\ list_1\ \textbf{in}\ (\underline{ident} \times \text{OBJECT})^+,\ \forall\ name_i\ \textbf{in}\ \underline{ident},\ \forall\ refdata_i\ \textbf{in}\ (\text{REF} \cup \text{DATA}),\\
\forall\ obj\text{: }\underline{object}(\underline{oid}(name),\ \underline{tuple}(list_2))\ \textbf{in}\ \text{SOBJECT},\\
has2Columns(list_2,\ [name_i],\ [refdata_i])\text{:}\\
\quad list_1 \times name \times (name_i \times (list_1 \rightarrow refdata_i))^+ \qquad \rightarrow obj \quad \textbf{class} \qquad _\#[_,_]
\end{array}$$

The operator **linkClass** takes a homogeneous sequence and transforms this sequence into a link class. The operands of **linkClass** are similar to the operands of the operator **class**. But in addition the names of two object types defining the source and the target nodes of the links must be provided:

$$\begin{array}{l}
\forall\ seq_1\text{: }\underline{seq}(list_1)\ \textbf{in}\ \text{SEQ},\ \forall\ source\ \textbf{in}\ \underline{ident},\ \forall\ target\ \textbf{in}\ \underline{ident},\\
\forall\ name_i\ \textbf{in}\ \underline{ident},\ \forall\ refdata_i\ \textbf{in}\ (\text{REF} \cup \text{DATA}),\\
\forall\ seq_2\text{: }\underline{seq}(<(name,\ \underline{linkObject}(\underline{oid}(name),\ ref_1,\ ref_2,\ \underline{tuple}(list_2)))>)\ \textbf{in}\ \text{SEQ},\\
\forall\ obj_1\text{: }\underline{object}(ref_1,\ _)\ \textbf{in}\ \text{SOBJECT},\ \forall\ obj_2\text{: }\underline{object}(ref_2,\ _)\ \textbf{in}\ \text{SOBJECT},\\
has2Columns(list_2,\ [name_i],\ [refdata_i]),\\
member(source,\ obj_1,\ list_1),\ member(target,\ obj_2,\ list_1)\text{:}\\
\quad seq_1 \times name \times source \times target \times (name_i \times (list_1 \rightarrow refdata_i))^+\\
\qquad\qquad\qquad\qquad \rightarrow seq_2 \qquad \textbf{linkClass} \qquad _\#[_,_,_,_]
\end{array}$$

Let us once more check whether all variables can be bound in this rather complex operator specification. From the operator application

$$seq_1\ \textbf{linkClass}[name,\ source,\ target,\ <(name_1, fun_1),\ ...,\ (name_n, fun_n)>]$$

the variables seq_1, *name*, *source*, *target*, $[name_i]$, and $[refdata_i]$ are bound (the $refdata_i$ are determined from the corresponding function terms). Variable seq_1 binds $list_1$ via the specification "$\forall\ seq_1$: $\underline{seq}(list_1)$ **in** SEQ". Then *source* and $list_1$ bind obj_1 through the member predicate; in the same way *target* and $list_1$ bind obj_2. Variables obj_1 and obj_2 bind ref_1 and ref_2, respectively, through their specifications. Through the *has2Columns* predicate, $[name_i]$ and $[refdata_i]$ bind $list_2$. Finally, *name*, ref_1, ref_2, and $list_2$ together bind seq_2 and the construction of the result type is complete. - Again there is also an object-manipulating version of the **linkClass** operator:

$$\begin{array}{l}
\forall\ list_1\ \textbf{in}\ (\underline{ident} \times \text{OBJECT})^+,\ \forall\ source\ \textbf{in}\ \underline{ident},\ \forall\ target\ \textbf{in}\ \underline{ident},\\
\forall\ name_i\ \textbf{in}\ \underline{ident},\ \forall\ refdata_i\ \textbf{in}\ (\text{REF} \cup \text{DATA}),\\
\forall\ obj\text{: }\underline{linkObject}(\underline{oid}(name),\ ref_1,\ ref_2,\ \underline{tuple}(list_2))\ \textbf{in}\ \text{LOBJECT},\\
\forall\ obj_1\text{: }\underline{object}(ref_1,\ _)\ \textbf{in}\ \text{SOBJECT},\ \forall\ obj_2\text{: }\underline{object}(ref_2,\ _)\ \textbf{in}\ \text{SOBJECT},\\
has2Columns(list_2,\ [name_i],\ [refdata_i]),\\
member(source,\ obj_1,\ list_1),\ member(target,\ obj_2,\ list_1)\text{:}\\
\quad list_1 \times name \times source \times target \times (name_i \times (list_1 \rightarrow refdata_i))^+\\
\qquad\qquad\qquad\qquad \rightarrow obj \qquad \textbf{linkClass} \qquad _\#[_,_,_,_]
\end{array}$$

Since all operators for accessing components of objects or tuples of objects have result types of kind OBJECT or DATA, we have to add an operator **ref** returning the object identifier of an object to be able to define mappings with result types of kind REF.

$\forall$ *obj*: <u>*abstractObject*</u>(*ref*,_) **in** OBJECT:			
obj	$\rightarrow$ *ref*	**ref**	#(_)

As an example for the application of the **class** operator we consider the query: List all direct connections from Dortmund to Munich. In the GraphDB-DML this query is expressed as:

```
on departure at_station station(s1), arrival at_station station(s2)
     departure in trip, arrival in trip
where s1.name = "Dortmund" and s2.name = "Munich"
derive dtime: departure.time, atime: arrival.time
```

We describe this query by the following algebra statements:

query
<departure, arrival, **rename**(station, s1), **rename**(station, s2), trip>
on[<**fun**(s) s **lspec**(departure, at_station, s1), **fun**(s) s **lspec**(arrival, at_station, s2),
fun(s) s **pspec**(departure, trip), **fun**(s) s **pspec**(arrival, trip)>]
select[**fun**(t) (t s1 name ="Dortmund") **and (**t s2 name ="Munich")]
class[table, <(dtime, **fun**(t) t departure time), (atime, **fun**(t) t arrival time)>]

Here for all variables the inferred type is <(departure, departure_type), (arrival, arrival_type), (s1, station_type), (s2, station_type), (trip, trip_type)>.

For the next example we assume that a class *state* has been defined as follows:

```
class state = sname: STRING; region: REGIONS;
```

The following query constructs a list of link objects connecting all states which are neighbours. Each object has an attribute storing the length of the common border.

```
on state(s1), state(s2)
where s1.region adjacent s2.region
derive neighbour_of = cblength: length(common_border(s1.region,
     s2.region)) from s1 to s2
```

In the GraphDB-algebra we describe the query by the following statements (where *state_type* is the type of *state* objects and *state* the name of the class, similar to definitions in Section 3.3):

query
<**rename**(state, s1), **rename**(state, s2)> **on**[<>]
select[**fun**(t) (t s1 region) **adjacent** (t s2 region)]
linkClass[neighbour_of, s1, s2,
<(cblength, **fun**(t) **length**(**common_border**(t s1 region, t s2 region)))>]

In this query, the operator **common_border** determines the line which describes the common part of the two regions given as operands. The operator **length** computes the length of a line.

4.3 Rewrite

The **rewrite** operator processes heterogeneous sequences of objects. In GraphDB heterogeneous sequences can be generated in three ways.

(1) A list of class names enclosed by angular brackets is a heterogeneous sequence. The sequence consists of all objects corresponding to classes whose names are contained in the list.
(2) The path corresponding to a path object can be viewed as a heterogeneous sequence of objects.
(3) Subgraphs which are returned by the graph operations of GraphDB are also viewed as heterogeneous sequences. These subgraphs are described by path types.

Before we discuss the **rewrite** operation in detail, we specify the types of heterogeneous sequences.

4.3.1 Heterogeneous Sequences

Heterogeneous sequences consist in general of arbitrary database objects whose order is described by a *sequence type*. A sequence type is described by a regular expression (or equivalently, a finite automaton) over object types. Path types defined in Section 3.2.3 are similar to sequence types but are restricted to sequences of objects corresponding to paths in the database graph. In contrast, there is no such restriction on sequence types; manipulation by the rewrite operator may transform a sequence of objects corresponding to a path type into an arbitrary sequence. In other words, a path type is a special case of a sequence type. A further technical difference is that a path type is described just in terms of link objects whereas a sequence type includes also the node objects of a path.

We introduce a kind HET and several type constructors for heterogeneous sequences:

OBJECT	→ HET	*elem*
HET^+	→ HET	*concat*, *or*
HET	→ HET	*star*, *plus*

The *elem* type constructor transforms any object type into a type for an element of a heterogeneous sequence (corresponding to a single symbol in a regular expression). *Concat* and *or* describe concatenation and union in regular expressions; *star* and *plus* the corresponding operators for regular expressions.

A heterogeneous sequence containing *book* and *person* objects in arbitrary order has a sequence type

star(*or*(<*elem*(*object*(*oid*(book), ...)), *elem*(*object*(*oid*(person), ...))>))

Such a sequence can be created in the GraphDB query language - for given classes *book* and *person* in the database - by writing

<book, person>

To model this in the algebra, we define an operator **hseq** creating a heterogeneous sequence from such a list. The notation in fact already describes a list consisting of two homogeneous sequences according to the definition in Section 4.2.2 (writing a class name generates a homogeneous sequence).

∀ *list* **in** SEQ^+, ∀ *het* **in** HET, *listHetType*(*list*, *het*):

list	→ *het*	**hseq**	_#

The predicate *listHetType* just computes from the given list of homogeneous sequence types the corresponding trivial regular expression which for object types t_1, ..., t_n is $(t_1 \cup ... \cup t_n)^*$ expressed in the algebra by a type

$\underline{star}(\underline{or}(<\underline{elem}(t_1), ..., \underline{elem}(t_n)>))$

There is a second operator **useq** converting a heterogeneous into a homogeneous ("uniform") sequence. This is merely a "type casting" operator; it just checks whether the regular expression of the heterogeneous sequence contains only a single object type in which case the sequence can be viewed as a homogeneous one. Checking is done by the predicate *oneType*.

$\forall$ *het* **in** HET, $\forall$ *seq*: $\underline{seq}$(<(*name*, *obj*)>) **in** SEQ, *oneType*(*het*, *name*, *obj*):
het → *seq* **useq** _#

4.3.2 The Rewrite Operation of GraphDB

We illustrate the **rewrite** operation of GraphDB by the following query which lists all direct connections from Dortmund to Munich with the distance travelled:

```
on departure at_station station (s1), arrival at_station station (s2),
     departure in trip, arrival in trip
where s1.name = "Dortmund" and s2.name = "Munich"
derive dtime: departure.time, atime: arrival.time, distance:
     trip suffix(departure) prefix(arrival)
     rewrite
        [departure ->, arrival ->, stay ->,
        travel -> travel_dist = (dist:
           travel.through.way
           rewrite[vertex ->,arc -> arc_length = len: length(arc.route)]
           sum[len])]
     sum[dist]
```

The tuple of objects resulting from the **on** and the **where-**clause describes all trips from Dortmund to Munich, the corresponding departure and arrival events and the objects describing Dortmund's and Munich's station. In the **derive** statement we determine the departure and the arrival time and store these values in attributes *dtime* and *atime*. To compute the distance traveled we access the part of the trip object describing the path between Dortmund and Munich by applying the operations **suffix** and **prefix**. These operations view each path object as a heterogeneous sequence. **Suffix** reduces the path (or any sequence) to the part after and including the object given as a second operand. **Prefix** reduces the path to the part before and including the object.

The remaining sequence is processed by a **rewrite** statement. This sequence consists of *departure*, *travel*, *arrival*, and *stay* objects. *Departure*, *arrival*, and *stay* objects are eliminated. *Travel* objects are transformed into new *travel_dist* objects with a single attribute *dist*. To compute this attribute, the physical route associated with the travel object via the *through* attribute, leading to a *connection* object, and then via a *way* attribute, is accessed. This path object is again processed by a **rewrite** statement. The corresponding sequence consists of *vertex* and *arc* objects. The *vertex* objects are eliminated and for each *arc* object the value of *arc_length* is computed describing the length of the arc. The sequence resulting from this second (nested) application of **rewrite** is viewed as a homogeneous sequence, since it consists of objects of a single object type. By the following application of the aggregate operator **sum** we compute the length of a single *travel.through.way* object. The final length of the way from Dortmund to Munich is obtained by computing the sum over all *dist* attributes of *travel_dist* objects.

The parameter list of the **rewrite** operation consists of a list of transformations. Each transformation consists of a pattern, which is a list of object types and an expression evaluating to an object type. The semantics of the **rewrite** operation of GraphDB is as follows. The operand sequence is processed in a series of steps. In each step a sequence pattern in one of the transformations is matched against a prefix of the operand sequence. The prefix is then removed and an object according to the expression in the transformation is put into the output sequence. To match the operand sequence with a pattern, the list of transformations is considered sequentially in the order of writing [Güti 94a]. If none of the pattern matches, an error is reported.

4.3.3 The Rewrite Operator

The **rewrite** operator of the GraphDB algebra takes a heterogeneous sequence of objects and a list of transformations as operands. These transformations are mappings transforming a subsequence matching the domain of the mapping into an object type.

$\forall$ het_1 **in** HET, $\forall$ $list_i$ **in** ($\underline{ident}$ $\times$ OBJECT)$^+$, $\forall$ obj_i **in** OBJECT, $\forall$ het_2 **in** HET,
transform(het_1, [$list_i$], [obj_i], het_2):
$het_1 \times (list_i \rightarrow obj_i)^+ \quad \rightarrow het_2$ **rewrite** _#[_]

The type of the heterogeneous sequence resulting from an application of **rewrite** is computed by the predicate *transform* which applies the transformations to the regular expression of the operand sequence to obtain a regular expression for the result sequence. For each mapping describing a transformation, the range must be an object type and the domain must be a tuple of (named) object types. The semantics of the **rewrite** operator of the GraphDB algebra is as described above for the **rewrite** operation of the GraphDB query language. If a mapping whose domain matches is defined by the operator **null**, the corresponding objects in the operand sequence are removed. This corresponds to patterns of the **rewrite** operation of the GraphDB query language having no expression on the right hand side of the arrow.

Note that an operand to a mapping - a tuple of named objects - has exactly the same form as in the **select** or **class** statement manipulating homogeneous sequences. This is needed because renaming of operands is also possible in a rewrite transformation. For example, suppose in a query we have generated in some way a sequence of *person* objects such that pairs of persons adjacent in the sequence are a married couple. A rewrite operator might then contain a transformation

person(husband) person(wife) -> couple = he: husband.name, she: wife.name

In the algebra this would be expressed by a mapping defined as

fun(x: <(husband, person), (wife, person)>) ...

Any subsequence of two person objects matches the domain of this mapping. On discovering such a subsequence, the rewrite operator passes a corresponding tuple of objects (whose first object is named “husband” and the second “wife”) to the corresponding function.

As an example for the application of the **rewrite** operator we now show the translation of the query above (top of next page). The operators **prefix**, **suffix**, **path**, **sum**, and **null** which are used are specified in Section 4.6.

```
query
    <rename(station, s1), rename(station, s2), departure, arrival, trip>
    on[<fun(s) s aspec(departure, at_station, s1),
        fun(s) s aspec(arrival, at_station, s2),
        fun(s) s pspec(departure, trip),
        fun(s) s pspec(arrival, trip)>]
    select[fun(t) (t s1 name = "Dortmund") and (t s2 name = "Munich")]
    class[table, <(dtime, fun(t) t departure time), (atime, fun(t) t arrival time),
        (distance,
        fun(t) t trip path suffix[t departure] prefix[t arrival]
            rewrite[<fun(d:<(departure, departure_type)>) null,
                fun(a:<(arrival, arrival_type)>) null,
                fun(s:<(stay, stay_type)>) null,
                fun(trav:<(travel, travel_type)>) trav
                    class[travel_dist, <(dist,
                        fun(u) u travel through way path
                            rewrite[<fun(v:<(vertex, vertex_type)>) null,
                                fun(a:<(arc, arc_type)>)
                                a class[arc_length, <(len, fun(v) length(v arc route))>]
                            useq
                            sum[fun(w) w arc_length len]
                    )>]
            >]
            useq
            sum[fun(x) x travel_dist dist]
    )>]
```

Here the inferred type of variables *s* and *t* (in the **on**, **select**, and outermost **class** operator) is

<(s1, station_type), (s2, station_type), (departure, departure_type),
(arrival, arrival_type), (trip, trip_type)>

The inferred type of *u* for the attribute function in the second **class** operator is <(travel, travel_type)> just as for the function containing it. This is the object-manipulating **class** operator which passes its operand tuple of objects unchanged to its attribute functions. In the same way, the inferred type of *v* is <(arc, arc_type)>. For the inner **sum** aggregate operator the inferred type of *w* is

<(arc_length, *object*(*oid*(arc_length), *tuple*(<(len, *integer*)>)))>

and for the outer aggregate function the type of *x* is

<(travel_dist, *object*(*oid*(travel_dist), *tuple*(<(dist, *integer*)>)))>

4.4 Union

To transform a heterogeneous sequence into a homogeneous sequence of objects the **union** operator is provided. **Union** takes a sequence of some type in HET and returns it as a homogeneous sequence. The actual objects in the sequence are not changed at all; they are only viewed under a common generalization type.

The tuple type of the resulting sequence is the smallest common supertype of all tuple types of the operand objects. The **union** operator produces homogeneous sequences containing the following kinds of objects [Güti 94a]:

- if the operand sequence contains simple objects only: simple objects,
- if the operand sequence contains link objects only and the target and source object types are related in the inheritance hierarchy: link objects,
- if the operand sequence contains link objects only and the target and source object types are not related in the inheritance hierarchy: simple objects,
- if the operand sequence contains path objects only: simple objects,
- if the operand sequence is a mixture of several kinds of objects: simple objects.

The specification of this operator is quite simple. Besides the heterogeneous sequence, a name must be provided as a second operand to describe the objects in the result sequence.

$\forall$ *het* **in** HET, $\forall$ *seq*: <u>*seq*</u>(<(*name*, *obj*)>) **in** SEQ, *superType*(*het*, *obj*):
het $\times$ *name* $\rightarrow$ *seq* **union** _#[_]

Of course, the complexity is hidden in the predicate *superType* which considers the object types occurring in the regular expression *het* to compute their common supertype *obj*. Note that all the information needed to realize this predicate is available: The data type hierarchy, needed to compute the smallest common supertype of any two data types in DATA, can simply be expressed explicitly in a predicate, e.g. by facts such as *subdata*(<u>*points*</u>, <u>*ext*</u>), etc. The class hierarchy needed to determine the smallest common super reference type (e.g. for <u>*oid*</u>(departure) and <u>*oid*</u>(arrival) it is <u>*oid*</u>(event)) can be accessed through the *isType* system predicate as we have seen already in Section 3.2.2. For a precise definition of the smallest common super type of any two tuple types see [Güti 94a].

4.5 Graph Operations

So far we only discussed operations processing sequences of objects. In this section we consider a limited set of operations on database graphs. Further research is required to define a complete set of graph operations for the GraphDB data model.

4.5.1 Shortest_path

The **shortest_path** operation computes a heterogeneous sequence of objects describing the shortest path in the database instance graph from a simple object to another simple object. The operation takes as operands:

- Two simple objects describing the source and target objects of the path.
- A path type describing the part of the complete data base graph to be considered. Since the **shortest_path** operator returns a heterogeneous sequence, this path type also defines the sequence type of the result.
- For each simple class in the relevant part of the data base graph a function giving an estimated distance from the current simple object (i.e. a node in the database graph) to the target node. This information is required for the A^* algorithm [Nils 82] which can be used to implement this operator.
- For each link class in the relevant part of the data base graph a function assigning a cost to a link object (i.e. an edge in the database graph).

As an example we consider the query: How many countries must be traversed traveling from Germany to China? In GraphDB this is expressed as follows [Güti 94a]:

```
on state(s1), state (s2)
where s1.region adjacent s2.region
derive neighbour_of from s1 to s2;

state(Germany) state(China)
     shortest_path[neighbour_of+,
          fun(n: neighbour_of) 1, fun(u: state, v: state) 0]
     rewrite[state ->, neighbour_of -> neighbour_of] count
```

In this case, trivial functions for edge cost and distance estimation are used which can in fact in the GraphDB query language be omitted and are then assumed as a default.

The basic problem in modeling the **shortest_path** operator is that it takes in principle a type parameter, namely the path type to be considered. However, terms of the algebra (of the bottom level algebra of a second-order signature!) do not manipulate types but only values. This is the same problem that we have encountered already in the definition of the **on** operator (whose list of subgraph specifications is essentially type information) and we use the same solution technique: We define a number of operators whose sole purpose is to represent path types.

The operator corresponding to the type constructor ***link*** is called **edge** and takes the name of a link class as an operand:

$\forall$ *path*: *link*(*oid*(*name*), *source*, *target*) **in** PATH:
name $\rightarrow$ *path* **edge** #(_)

The operator corresponding to the *concat* type constructor is the & operator. It takes two instances of path types as operands:

$\forall$ *path*: *concat*($path_1$, $path_2$) **in** PATH:
$path_1 \times path_2 \rightarrow path$ & _#_

The operator **|** describes instances of types defined by the *or* type constructor:

$\forall$ *path*: *or*($path_1$, $path_2$) **in** PATH:
$path_1 \times path_2 \rightarrow path$ **|** _#_

The operator corresponding to the type constructor *plus* is the postfix +-operator:

$\forall$ $path_2$: *plus*($path_1$) **in** PATH
$path_1 \rightarrow path_2$ + _#

Finally, the operator * corresponds to types which are constructed by the *star* type constructor:

$\forall$ $path_2$: *star*($path_1$) **in** PATH
$path_1 \rightarrow path_2$ * _#

Now we specify the **shortest_path** operator. It takes five operands: Two simple objects which describe the start and the end node of the path, an instance of a path type, and the two lists of mappings mentioned above.

$\forall$ *source* **in** SOBJECT, $\forall$ *target* **in** SOBJECT, $\forall$ *path* **in** PATH,
$\forall$ $edge_i$ **in** LOBJECT, $\forall$ $node_j$ **in** SOBJECT, $\forall$ *het* **in** HET,
pathHetType(*path*, *het*),
allEdgesDefined(*het*, [$edge_i$]),
allNodesDefined(*het*, [$node_j$]):
$source \times target \times path \times (edge_i \rightarrow \underline{real})^+ \times (node_j \times target \rightarrow \underline{real})^+$
$\rightarrow het$ **shortest_path** _ _#[_,_,_]

The predicate *pathHetType* computes the sequence type *het* from the given path type *path*; essentially it inserts the traversed node types. The predicates *allNodesDefined* and *allEdgesDefined* check whether all node and edge types used in the sequence type *het* do occur as domains of functions in the corresponding lists.

The algebraic representation of the first part of the query above has already been shown in Section 4.2.5. The second part of the query can be expressed in the GraphDB algebra as follows:

type neighbour_of_type =
  <u>*linkObject*</u>(<u>*oid*</u>(neighbour_of), <u>*oid*</u>(state), <u>*oid*</u>(state), <u>*tuple*</u>(<>))
query
  state **object**[**fun**(t) t sname = "Germany"]
  state **object**[**fun**(t) t sname = "China"]
  shortest_path[**edge**(neighbour_of) +,
    <**fun**(n: neighbour_of) 1>,
    <**fun**(u: state, v: state) 0>]
  rewrite[<**fun**(s: <(state, state_type)>) **null**,
    fun (n: <(neighbour_of, neighbour_of_type)>) n neighbour_of>]
  count

4.5.2 Circle

In GraphDB the **circle** operation determines a subgraph within some radius from a given start node. Again we define a corresponding graph operator **circle**. This operator takes the following operands:

- A simple object denoting the start node.
- An instance of a path type describing the part of the complete data base graph to be considered.
- A function assigning a cost to each edge that may occur in the path.
- The radius of the circle.

We specify the operator **circle** as follows:

$\forall$ *source* **in** SOBJECT, $\forall$ *path* **in** PATH, $\forall$ $edge_i$ **in** LOBJECT, $\forall$ *het* **in** HET,
pathUnorderedHetType(*path*, *het*), *allEdgesDefined*(*het*, [$edge_i$]):
  $source \times path \times (edge_i \rightarrow \underline{real})^+ \times \underline{real} \quad \rightarrow het$ **circle**   _ #[_,_,_]

The definition is quite similar to that of the **shortest_path** operator. However, since this operator traverses many paths simultaneously, enumerating objects in increasing distance from the start node, the resulting heterogeneous sequence may contain these objects in some arbitrary order. The predicate *pathUnorderedHetType* therefore constructs a sequence type *het* which allows any order for the objects occurring in *path*, like the sequence types constructed by predicate *listHetType* for the operator **hseq**.

4.6 Other Operators

We now specify other operators which are useful for querying the database. Some of the operators presented in this section have already been used in preceding examples.

4.6.1 Operators on Paths and Heterogeneous Sequences

The operator **path** is required to extract the path corresponding to a path object. This path is returned as a heterogeneous sequence of objects:

$\forall$ *pobj*: <u>*pathObject*</u>(_ , *path*, _) **in** POBJECT, $\forall$ *het* **in** HET, *pathHetType*(*path*, *het*):
pobj $\rightarrow$ *het* **path** _ #

The operator **start** determines the first object of the path corresponding to a path object:

$\forall$ *pobj*: <u>*pathObject*</u>(_ , *path*, _) **in** POBJECT, $\forall$ *obj* **in** SOBJECT, *first*(*obj*, *path*):
pobj $\rightarrow$ *obj* **start** _ #

The predicate *first* determines the simple object type *obj* as the type of the first node in the path described by the path type *path*.

The operator **end** determines the last object of the path corresponding to a path object:

$\forall$ *pobj*: <u>*pathObject*</u>(_ , *path*, _) **in** POBJECT, $\forall$ *obj* **in** SOBJECT, *last*(*obj*, *path*):
pobj $\rightarrow$ *obj* **end** _ #

The predicate *last* tests whether the simple object type *obj* is the type of the last node in the path described by the path type *path*. If the path *path* denotes a path with several final nodes, the test fails.

The operators **prefix** and **suffix** are applied to a heterogeneous sequence and take an object as an additional operand. **prefix** reduces the sequence to the part before and including the object. **suffix** reduces the path to the part after and including the object:

$\forall$ het_1 **in** HET, $\forall$ *obj* **in** OBJECT, $\forall$ het_2 **in** HET, *prefixType*(het_1, *obj*, het_2):
$het_1 \times obj$ $\rightarrow het_2$ **prefix** _#[_]

$\forall$ het_1 **in** HET, $\forall$ *obj* **in** OBJECT, $\forall$ het_2 **in** HET, *suffixType*(het_1, *obj*, het_2):
$het_1 \times obj$ $\rightarrow het_2$ **suffix** _#[_]

Here the predicates *prefixType* and *suffixType*, respectively, compute the sequence types of the remaining parts of the sequence. This is not trivial, but possible. The strategy for determining the result types is the following. The object type of the parameter object *obj* may in general occur in several positions of the regular expression het_1 describing the argument sequence type. It is impossible to know at type checking time at which of these possible positions the parameter object will be found. Hence the result type will be the union of the regular expressions corresponding to finding the object at any of these positions.

As an example, consider the application of the **suffix** operator in the query of Section 4.3.2

```
trip suffix(departure)
```

which computes the remainder of a *trip* after a specific *departure* object (the one occurring at station “Dortmund”). The path type of trip is “travel (stay travel)*” which translates into a sequence type

D T A (S D T A)*

(D = departure, T = travel, A = arrival, S = stay). The departure object may be found in two positions, namely

D T A (S D T A)* or D T A (S **D** T A)*

For each of those, we consider the regular expression of the remaining sequence, and form their union:

D T A (S D T A)* ∪ D T A (S D T A)* = D T A (S D T A)*

As it happens, the types of the remaining sequence are the same for both possible positions of the departure object. Hence in this case the result type is the same as the argument type.

The operator **null** is used to construct special functions for the **rewrite** operator. **Null** returns a simple object whose tuple part does not contain any attributes:

∀ *obj*: <u>*object*</u>(<u>*oid*</u>(null), <u>*tuple*</u>(<>)) **in** SOBJECT:

	→ *obj*	**null**	#

Whenever the **rewrite** operator receives objects of this special type, it omits them, i.e. writes nothing into the result sequence.

4.6.2 Aggregate Operators

We now specify some aggregate operators. **count** processes arbitrary homogeneous or heterogeneous sequences and returns the number of objects in the sequence:

∀ *seq* **in** (SEQ ∪ HET):

seq	→ <u>*integer*</u>	**count**	_#

Operators **sum**, **min**, **max**, and **avg** can only be applied to homogeneous sequences. **sum** adds the values returned by the second operand which is a mapping from the tuple of object types underlying the homogeneous sequence to <u>*integer*</u> or <u>*real*</u>. **min**, **max** and **avg** compute the minimum, the maximum and the average of the values returned by the second operand, respectively.

∀ *seq*: <u>*seq*</u>(*list*) **in** SEQ, ∀ *num* **in** NUM:

seq × (*list* → *num*)	→ *num*	**sum**, **min**, **max**, **avg**	_#[_]

4.6.3 Operators for Atomic Data Types

Finally, we show the specifications for the operators on atomic data types. Values of atomic types and object identifiers can be compared.

∀ *refdata* **in** REF ∪ DATA:

refdata × *refdata*	→ <u>*bool*</u>	=, <>	_#_

Additionally numbers and strings can be ordered. That means that the following additional comparisons are defined for these types.

<u>*num*</u> × <u>*num*</u>	→ <u>*bool*</u>	<, >, ≥, ≤	_#_
<u>*string*</u> × <u>*string*</u>	→ <u>*bool*</u>	<, >, ≥, ≤	_#_

For the geometric types various comparison operators can be defined. As an example we show the specification of the operator **adjacent** which tests whether two regions are adjacent.

<u>*regions*</u> × <u>*regions*</u>	→ <u>*bool*</u>	**adjacent**	_#_

For the geometric data types various operators returning information about the shape of the geometric objects can be defined. We include the specification of the operators **length** and

common_border. **length** takes a geometric object of type *lines* and returns a value of type *real*.

lines	→ *real*	**length**	**_#_**

common_border takes two geometric objects of type *regions* and returns the common border of both objects as a geometric object of type *lines*:

regions × *regions*	→ *lines*	**common_border**	#(_,_)

The geometric operations are part of the ROSE algebra [GüSc 93] which is itself specified as a second-order signature.

5 Related Work

The specification framework of second-order signature combines in a particular way the notions of signature with the use of type parameters for the specification of type systems and polymorphic operations. The use of type parameters is known as parametric polymorphism; the formal basis for this is second-order typed lambda calculus. Cardelli and Wegner [CaWe 85] give a survey of types and polymorphism in programming languages. Second-order signature uses a three level framework consisting of *kinds* (the "types" of types), *types*, and *values*. Such a three-level type structure was also proposed by Cardelli et al. [Card 88, CaLo 90], see [Güti 93] for a detailed discussion of differences. Research in database programming languages has also addressed the problem of achieving a precise typing for database queries (e.g. [Stem 90, OBB 89, BuOh 91]). A main goal is to make static type checking possible. For example, in the language Machiavelli [OBB 89] projection and join operations can be defined and the result types of operations be inferred. A general distinction between programming language type systems and the SOS framework is that SOS has an "external semantic component" reflected in the separation between second-order signature and second-order algebra. This is motivated by the purpose of SOS to model extensible systems which basically manage collections of externally defined data structures (for the type constructors) and procedures (for the operations). The SOS parser & optimizer as well as the SOS parser & execution system components actually do not need to know the semantics of type constructors and operations; they work just with the signature and the optimization rules. In contrast, in PL type systems, the semantics are always included in the definition of entities such as types, polymorphic functions, type operators, etc.

Graph-based data models which may be compared to GraphDB have been proposed, for example, by [CMW 87, Agra 87, BRS 90, ErGü 94, GPVV 94, KSW 95]. Since the purpose of this paper is not to propose a data model but rather to offer an SOS specification of the given model GraphDB, we do not discuss these models here, see [Güti 94a, Güti 94b].

Object-oriented query algebras can be found, for example, in [ScSc 90, ShZd 90, StÖz 90, VaDe 91]. Two major differences to the GraphDB algebra are:

- The type systems for these algebras are defined in the usual mathematical (ad-hoc) notation; the GraphDB algebra is the only one described within a formal framework for defining type systems. The experience with GraphDB shows that these algebras could probably also be defined without much difficulty in the SOS framework.
- These algebras lack concepts for the graph-specific aspects of GraphDB, for example, the treatment of link classes, path types, and path classes in the data model, the specification of relevant subgraphs (in the **on** operator) or generation of link objects in querying. Also the part of the GraphDB algebra offering a sophisticated manipulation of heterogeneous sequences has no counterpart in these algebras.

Some similarities and differences in detail are the following. Similar to [ShZd 90] the GraphDB algebra offers object preserving and object generating operators whereas [ScSc 90] and [StÖz 90] only define object preserving operators. The **on** operator generates new tuples of objects, but these objects are already present in the database. Hence, the **on**-operator is a generalization of the **ojoin** operator of [ShZd 90] or the object generating **join** operator of [ScSc 90] which only combine pairs of objects. The object preserving join in [ScSc 90] is realized by an **extend** operator which adds new attributes or methods to objects. This operation cannot be performed with the GraphDB-algebra. Using the **class** or the **linkClass** operator an object generating variant of the **extend** operator could be defined. The **select** operator of the GraphDB-algebra is similar to the **select** operations of [ScSc 90] and [ShZd 90]. Since it processes sequences of tuples of objects, we can use this operator like the **select** operator of [StÖz 90] to compute joins. In this case the **on** operator is required to combine the operands of the join. The **class** and **linkClass** operators generate new objects. These operators are similar to the **image** and **project** operators of [ShZd 90].

The approach taken in [VaDe 91] is different from [ScSc 90], [ShZd 90], [StÖz 90], and our algebra. [VaDe 91] define orthogonal type constructors for multiset-types, tuple-types, array-types, and object identifiers. Instead of defining most operators to work on sets of entities they provide operators for each of the four basic types.

The GraphDB data model does not associate methods with objects; hence these do not occur in our algebra either. In contrast, other object algebras like [ScSc 90], [ShZd 90], [StÖz 90], and [VaDe 91] include a treatment of methods.

6 Conclusions

We have presented a specification for the type system and a specification for a query algebra of the graph-based data model GraphDB [Güti 94a]. The formal base for this specification are second-order signatures as defined in [Güti 93]. In [Güti 93] a specification for the relational data model has been presented. By the specification of the type system and the query algebra of GraphDB we have shown that second-order signature is also suitable for the specification of complex data models. Future work includes:

- The SECONDO parser & execution system mentioned in the introduction has already been implemented [Ende 94] reflecting the specification technique as of [Güti 93]. It needs to be adapted to the extensions of the technique such as user-defined predicates and database quantification.
- The SOS parser & optimizer needs to be designed and implemented. Optimization strategy and rule format can perhaps be adapted from [BeGü 92]. An example optimization rule in the SOS framework can be found in [Güti 93].
- A complete formal definition of syntax and semantics of SOS specifications is still missing. In [Güti 93] only a formal core has been defined.

References

[Agra 87] Agrawal, R., ALPHA: An Extension of Relational Algebra to Express a Class of Recursive Queries. Proc. IEEE Data Engineering Conf. 1987, 580-590.

[BeGü 92] Becker, L., and R.H. Güting, Rule-Based Optimization and Query Processing in an Extensible Geometric Database System, *ACM Transactions on Database Systems 17 (1992)*, 247 - 303.

[BuOh 91] Buneman, P., and A. Ohori, A Type System that Reconciles Classes and Extents. Proc. 3rd Int. Workshop on Database Programming Languages (Nafplion, Greece), 1991, 191-202.

[BRS 90] Biskup, J., U. Räsch, and H. Stiefeling, An Extension of SQL for Querying Graph Relations. *Computer Languages 15* (1990), 65-82.

[Card 88] Cardelli, L., Types for Data-Oriented Languages. Proc. EDBT (LNCS 303), 1988, 1-15.

[CaLo 90] Cardelli, L., A Semantic Basis for Quest. Proc. ACM Conf. on LISP and Functional Programming, 1990, 30-43.

[CaWe 85] Cardelli, L., and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys 17 (1985)*, 471-522.

[CMW 87] Cruz, I.F., A.O. Mendelzon, and P.T. Wood, A Graphical Query Language Supporting Recursion. Proc. ACM SIGMOD 1987, 323 - 330.

[Ende 94] Endemann, M., SECONDO: Ein Parser und Ausführungssystem für SOS-Datenbanken. Fernuniversität Hagen, Diplomarbeit (Master Thesis), 1994.

[ErGü 94] Erwig, M., and R.H. Güting, Explicit Graphs in a Functional Model for Spatial Databases. *IEEE Transactions on Knowledge and Data Engineering 6:4 (1994)*, 572-586.

[Grae 94] Graefe, G., Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering 6:1 (1994)*, 120-135.

[GrDe 87] Graefe, G., and D. J. DeWitt, The EXODUS Optimizer Generator. Proc. ACM SIGMOD 1987.

[GrMc 93] Graefe, G., and W.J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search. Proc. IEEE Intl. Conf. on Data Engineering (Vienna, Austria), 1993, 209-218.

[Güti 93] Güting, R.H., Second-Order Signature: A Tool for Specifying DataModels, Query Processing and Optimization. Proc. ACM SIGMOD 1993, 277 - 286.

[Güti 94a] Güting, R. H., GraphDB: A Data Model and Query Language for Graphs in Databases. Fernuniversität Hagen, Informatik-Report 155, 1994.

[Güti 94b] Güting, R.H., GraphDB: Modeling and Querying Graphs in Databases. Proc. of the 20th Intl. Conf. on Very Large Data Bases, 1994, 297-308.

[GüSc 93] Güting, R.H., and M. Schneider, Realm-Based Spatial Data Types: The ROSE Algebra. Fernuniversität Hagen, Informatik-Report 141, 1993, to appear in the *VLDB Journal.*

[GPVV 94] Gyssens, M., J. Paredaens, J. Van den Bussche, and D. Van Gucht, A Graph-Oriented Object Database Model. *IEEE Transactions on Knowledge and Data Engineering 6:4 (1994)*, 572-586.

[Haas 90] Haas, L. M., et al., Starburst Mid-Flight: As the Durst Clears. *IEEE Transactions on Knowledge and Data Engineering 2:1 (1990).*

[KSW 95] Kiesel, N., A. Schürr, and B. Westfechtel, GRAS: A Graph-Oriented (Software) Engineering Database System. *Information Systems 20:1 (1995)*, 21-51.

[Nils 82] N.J. Nilsson: Principles of Artificial Intelligence. Springer-Verlag, Berlin, 1982.

[OBB 89] Ohori, A., P. Buneman, and V. Breazu-Tannen, Database Programming in Machiavelli - a Polymorphic Language with Static Type Inference. Proc. ACM SIGMOD, 1989, 46-57.

[ScSc 90] Scholl, M.H., and H.J. Schek, A Relational Object Model. Proc. Int. Conf. on Database Theory, Paris, France, 1990, 89-105.

[ShZd 90] Shaw, G.M., and S.B. Zdonik, A Query Algebra for Object-Oriented Databases. Proc. IEEE Intl. Conf. on Data Engineering, 1990, 154 - 162.

[Stem 90] Stemple, D., L. Fegaras, T. Sheard, and A. Socorro, Exceeding the Limits of Polymorphism in Database Programming Languages. Proc. EDBT (LNCS 416), 1990, 269-285.

[StÖz 90] Straube, D.D., and M.T. Özsu, Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems 8 (1990)*, 387 - 430.

[SPSW 90] Schek, H.J., H.B. Paul, M.H. Scholl, and G. Weikum, The DASDBS Project: Objectives, Experiences, and Future Prospects. *IEEE Transactions on Knowledge and Data Engineering 2 (1990)*, 25-43.

[SRH 90] Stonebraker, M., L.A. Rowe, and M. Hirohama, The Implementation of Postgres, *IEEE Transactions on Knowledge and Data Engineering 2:1 (1990).*

[VaDe 91] Vandenberg, S.L., and D.J. DeWitt, Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. Proc. ACM SIGMOD 1991, 158-167.