Second-Order Signature: A Tool for Specifying Data Models, Query Processing, and Optimization

Ralf Hartmut Güting

Praktische Informatik IV, FernUniversität Hagen
Postfach 940, D-5800 Hagen, Germany
gueting@fernuni-hagen.de

and

Institut für Informationssysteme
Departement Informatik, ETH Zürich, Switzerland

Abstract: We propose a framework for the specification of extensible database systems. A particular goal is to implement a software component for parsing and rule-based optimization that can be used with widely varying data models and query languages as well as representation and query processing systems. The key idea is to use *second-order signature* (and algebra), a system of two coupled many-sorted signatures, where the top-level signature offers kinds and type constructors and the bottom-level signature provides polymorphic operations over the types defined as terms of the top level. Hence the top level can be used to define a data or representation model and the bottom level to describe a query algebra or a query processing algebra. We show the applicability of this framework by examples drawn from relational modeling and query processing.

Keywords: Type system, algebra, signature, polymorphism, functional programming, extensibility, data model, query processing, optimization

---

This work was done during a sabbatical stay at the Institut für Informationssysteme, ETH Zürich

1 Introduction

Extensible database systems have now been studied for almost a decade. An extensible DBMS provides, first of all, support for adding data types and their operations ("abstract data type support"). This simple requirement affects system architecture at all levels: It must be possible not only to add representations for the types and procedures for the operations, but also special types of possibly clustering index structures (e.g. for geometric search), new query processing methods (such as special join algorithms), and extensions of the optimizer (e.g. by rules) to map data type operations in the query language to the use of the new index structures and query processing algorithms at the executable level. Quite a few extensible systems have been built and a lot of progress has been made on the engineering side (e.g. [StR86, GrD87, Bato88, Haas89, Sche90]). However, most of the systems lack a clear formal framework to define what extensibility means. Which kinds of data models can be supported? Which additions are possible for representation and query processing? This is usually described in an ad-hoc manner, at best using many different formalisms at various levels of an extensible system.

The goal of our work is to provide a comprehensive framework with a clean formal basis to describe data models and query languages, representation models and execution languages, and optimization rules to map between these two levels. Some initial work in this direction has been reported [Gü89] and been exemplified in a prototype, the Gral system. There, the formal basis is *many-sorted algebra*. Such an algebra defines an application-specific query language, another algebra a representation system with special kinds of index structures and query processing algorithms. Optimization rules are formulated as transformations of algebra terms with variables and an optimizer can be built as a rule file consisting of several steps where each step has its own collection of rules and employs a selected control strategy [BeG92].

However, the underlying framework in Gral is still restricted in several ways. Gral is an extensible *relational* system, so in the data model an arbitrary collection of atomic data types is allowed, but the only structures are relations. At the representation level a fixed assumption is that there are relation representations (clustering structures) and index structures. The many-sorted algebra framework is only used to describe queries but not updates; updates are hard-wired into the system, translating relational updates to changes in relation representations and indexes.

We now pursue the much more ambitious goal of providing a framework so general that it is possible

- to define a *data model* or *type system* together with an *algebraic* (or functional) *query language*. For example, it should be possible to define the relational model, nested relational models, complex object models, object-oriented models, or graph models (e.g. [ScS91, CrMW87, GyPV90]) with corresponding algebraic query languages.
- to describe *representations* and *query processing* as another type system and algebra. For example, one should be able to model clustering representations for classes of objects, special kinds of index structures, graph-like representations and so forth together with algebraic execution languages using these structures.
- to write *optimization rules* as rewrite rules on terms of these algebras.
- to treat *updates* within the same framework. In the data model as well as in the representation model, updates will be described as algebraic operations. There will be optimization rules to translate model updates into representation updates.

- to describe the *database catalog* also as an algebraic structure. The database catalog cannot be hard-wired when data model as well as representation model are allowed to vary.

It should be clear now that we are planning to develop some kind of “meta-model”: a model wherein data models can be defined. This is somewhat analogous to developing formal grammars as a means to describe the syntax of programming languages. The *practical benefits* are in that example parts of compiler technology up to compiler generation tools. In our case the practical benefits will be a clean extensible system architecture and in particular an extensible parser and optimizer software component independent of any specific data model or representation model. One will be able to write a concise specification as data for that component; after that the component will be able to accept programs consisting of data definitions, data manipulation and queries with respect to the specified data model and to translate it into corresponding programs for the representation model.

Our approach to provide such a framework is to introduce the concept of a *second-order signature* (and algebra) that we define to consist of two coupled many-sorted signatures. The first signature defines a *type system*; its operators are *type constructors* and its terms are *types*. The second signature uses the terms of the first signature as sorts over which operators (of query language or execution language) are defined. Second-order signatures are the formal backbone; one writes then *specifications*, in particular of polymorphic operations, that translate into a second-order signature (and so possess formal semantics). Additionally, a very simple generic data definition and manipulation language is defined. The same language is used at the model level and at the representation level.

Clearly, the goal is ambitious and we are still far from a complete solution. This paper is meant to expose and illustrate the basic idea and to lay the ground for further work. We explain the framework informally in Section 2, giving examples of data model specifications. Second-order signature as the formal core is defined in Section 3. Section 4 shows example specifications at the representation or query processing level. The formulation of optimization rules and updates is demonstrated in Sections 5 and 6. Finally, we discuss related work and some important features of our approach in Section 7.

The idea of using a two-level algebraic framework where the top-level algebra describes a type system and the bottom level an algebra over these types was developed in joint work with Martin Erwig. In [ErG91] a preliminary description of such a framework was given and applied to define a data model that integrates object class hierarchies with explicit graph structures.

2 The Framework: Informal Introduction

We start from the concept of a *signature* which consists of two sets of symbols called *sorts* and *operators*; operators are annotated with strings of sorts. Signatures are well-known from the specification of abstract data types, for example:

sorts	*stack, int, bool*		
ops	$stack \times int$	$\rightarrow stack$	*push*
	stack	$\rightarrow int$	*top*
		$\rightarrow stack$	*empty*

To assign semantics to a signature, one must assign a set to each sort and a function to each operator which has domains and codomain according to the string of sorts of the operator. Such a collection of

sets and functions forms a (many-sorted) *algebra*. A signature describes the syntactic aspect of an algebra by defining a set of *terms* such as *top*(*push*(*empty*, 7)). The *sort of a term* is the result sort of its outermost operator.

2.1 Specifying a Type System

The idea is now to use a signature to define a *type system* which in turn describes the structural part of a data model or a representation model. *Operators* of the signature play the role of *type constructors* (constant operators such as *empty* above denote constant types here) and *terms* denote *types. Sorts* are called *kinds* here. Since the set of terms is partitioned by result sorts, each *kind* stands for a *set of types*. – For example, let us specify the structural part of the relational data model in this way:

kinds IDENT, DATA, TUPLE, REL
type constructors

	→ IDENT	*ident*
	→ DATA	*int*, *real*, *string*, *bool*
(*ident* × DATA)$^+$	→ TUPLE	*tuple*
TUPLE	→ REL	*rel*

The only type (term) of kind IDENT is the constant type *ident* describing a domain of identifiers (which can be used as attribute names). Similarly DATA contains only some constant types. Using the equality symbol to associate a kind with its set of types, we may write IDENT = {*ident*}, DATA = {*int*, *real*, *string*, *bool*}. In contrast, TUPLE and REL contain an infinite number of types. For example,

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

The definition of the *tuple* type constructor uses already some extensions to the basic concepts mentioned before. The first is that a type constructor may use as sorts not only kinds but also types. The second extension makes it possible to define operators taking a variable number of operands. The notation s^+ denotes a list of one or more operands of sort s. The third extension is that if $s_1, \ldots, s_n$ are sorts, then $(s_1 \times \ldots \times s_n)$ is also a sort. Hence (*ident* × DATA)$^+$ is a sort; a term of that sort is a list of pairs where the first component is a value of type *ident* and the second a type of kind DATA.

Note that the notion of a (relation) *schema* is absent and has been replaced by a (relation) *type*. Hence "relation" is viewed as a type constructor leading to many different types; it is not a single type. This means that relational operations such as selection or join will be viewed as polymorphic operations.

Note further that the choice of kinds and type constructors is completely left to the designer; there is no fixed set of constructors such as *set*, *tuple*, *list*, etc. So we are not offering a toolbox of constructors that may be used to specify a data model but rather a framework to define any constructors that might be necessary. The presence of kinds allows one to control precisely how constructors can be applied.

To show that this is a general framework let us briefly consider type systems for nested relations and for complex objects. Nested relations can be modeled as follows:

kinds IDENT, DATA, REL
type constructors

	→ IDENT	*ident*
	→ DATA	*int*, *real*, *string*, *bool*
(*ident* × (DATA ∪ REL))$^+$	→ REL	*rel*

Here we have used another extension of the concept of signature: If s_1, …, s_n are sorts, then ($s_1 \cup \ldots \cup s_n$) is also a sort. A type in REL for representing books is, for example:

rel(< (title, *string*),
(authors, *rel*(< (name, *string*),
(country, *string*) >)),
(publisher, *string*),
(year, *int*) >)

Here is a complex object type system in the spirit of [BaK86]:

kinds IDENT, OBJ
type constructors

	→ IDENT	*ident*
	→ OBJ	*bottom*, *top*, *int*, *real*, *string*, *bool*
(*ident* × OBJ)$^+$	→ OBJ	*tuple*
OBJ	→ OBJ	*set*

A type in OBJ to represent persons is:

tuple(< (name, *string*),
(children, *set*(*string*),
(address, *tuple*(< (city, *string*),
(street, *string*) >) >)

2.2 Specifying Operations

Once a type system has been defined one can specify operations on those types. The presence of kinds provides an excellent basis to specify polymorphic operations precisely, since *quantification over kinds* can be used. We consider some operations for the relational model given above.

∀*data* **in** DATA. *data* × *data* → *bool* =, ≠, <, ≤, ≥, >

Here *data* is a variable ranging over the types (terms) in kind DATA. The idea is that a choice is made and the chosen type is substituted into the remainder of the specification. For each type that can be substituted we obtain one functionality for each of the operators listed. Hence the comparison operators can be applied to two integers, two strings, etc.

∀*rel*: *rel*(*tuple*) **in** REL. *rel* × (*tuple* → *bool*) → *rel* **select**

This describes relational selection. Here *rel*(*tuple)* is a *pattern* in the quantification used to bind the two variables *rel* and *tuple* simultaneously. The variable *rel* can be bound to any type in REL that has *rel* as the outermost constructor. Binding *rel* assigns the corresponding tuple type to the variable *tuple*. Hence the **select** operator so defined takes two operands, namely a relation and a function evaluating

for a tuple of the relation to *true* or *false*. The specification also expresses that the type (schema) of the resulting relation is the same as that of the operand relation. Note however, that the definition is purely syntactical: Nothing is said about how the polymorphic function **select** determines the result relation. This is only fixed when an algebra is associated with the signature.

This specification also shows the last extension of the concept of signature that we use: If for $n \geq 0$, s_1, …, s_n and s are sorts, then $(s_1 \times \ldots \times s_n \rightarrow s)$ is also a sort. All these extensions are formally defined below.

You may wonder now how one can access the components of a tuple (attributes) in order to define functions serving as predicates on tuples. Here is the specification of tuple access:

$\forall$*tuple*: *tuple*(*list*) **in** TUPLE. $\forall$(*attrname*, *dtype*) **in** *list*.

tuple $\rightarrow$ *dtype* *attrname*

The first quantification ranges over tuple types, binds *tuple* to such a type and *list* to the particular list of pairs of identifiers and DATA types used in that tuple type. The second quantification ranges over that list and, for each pair in the list, binds the variables *attrname* and *dtype*. For each binding, an operator *attrname* is defined that returns for a *tuple* value a *dtype* value.

Here is a relational union operator that takes a variable number of operands (one or more). Note how the usually separately given constraint that union is only possible for relations with the same schema is expressed concisely in this specification.

$\forall$*rel* **in** REL. $rel^+ \rightarrow rel$ **union**

Finally, a binary join operator can be specified as follows:

$\forall rel_1$: *rel*($tuple_1$) in REL. $\forall rel_2$: *rel*($tuple_2$) in REL.

$rel_1 \times rel_2 \times (tuple_1 \times tuple_2 \rightarrow bool) \rightarrow rel$: REL **join**

The only new concept here is the way the result type of the join is specified. We recall that the result schema of a relational join is the concatenation of the two operand schemas (or the union of the two attribute sets). In the type system defined here, a relation schema is really given by the tuple type to which the *rel* constructor is applied. So here the result type should be *rel* applied to the concatenation of the two tuple types. How is that expressed here?

The answer is that it is not expressed because it is part of the semantics of the **join** operator. We have so far seen that in the framework there are *kinds*, *type constructors*, *types* (as the terms built from type constructors), and *operators* defined on types. Indeed, there is one more concept: *type operators*. A type operator is (usually) defined on kinds and it maps operand types of the respective kinds into a type of the result kind. The specification given above, when translated into a second order signature (defined below) which represents its semantics, includes the definition of a type operator:

REL $\times$ REL $\rightarrow$ REL **join**

So this type operator computes for two given REL types the result type in REL. How it does that is again left to the stage when an algebra is associated with the second order signature. Hence the specification above introduces simultaneously under the name of **join** one *type operator* and very many *operators* (with distinct functionalities). Alternatively, the latter may be viewed as one

polymorphic operator. We read the notation *rel*: REL as “some type in REL” where it is understood that this type is determined by the corresponding type operator. In Section 7 the need for type operators is discussed further.

2.3 Syntax

The standard syntax for terms over a signature is prefix notation, that is, one writes first the operator and then, in parentheses, the operands. We extend this by some rules for the extensions of signature we have made such as list sorts, product sorts, union sorts, and function sorts. For example, a term of sort s^+ is written as $<t_1, \ldots, t_n>$ where the t_i are terms of sort s. The resulting syntax for terms is called the *abstract syntax*. It is used in all formal definitions.

On the other hand, the algebras specified are also to serve as readable query languages (without syntactic sugar but still with a somewhat pleasing appearance). To this end we allow to specify a *syntax pattern* for each operator which, together with some slightly relaxing rules for writing operand lists etc., leads to a *concrete syntax*. For example, the following syntax patterns might have been defined for the operators introduced so far:

$data \times data$	$\rightarrow \underline{bool}$	$=, \neq, <, \leq, \geq, >$	(_ # _)
$rel \times (tuple \rightarrow \underline{bool})$	$\rightarrow rel$	**select**	_ #[_]
$tuple$	$\rightarrow dtype$	*attrname*	_ #
rel^+	$\rightarrow rel$	**union**	_ #
$rel_1 \times rel_2 \times (tuple_1 \times tuple_2 \rightarrow \underline{bool})$	$\rightarrow rel$: REL	**join**	_ _ #[_]

In these patterns “_” denotes an operand and “#” the operator; square and round brackets have to be put as in the pattern. If no syntax pattern is given, prefix notation is assumed as a default.

Before we can show an example of concrete syntax we need to introduce the notation for terms of function sorts (which is the same in abstract and concrete syntax). That is, one must be able to denote values that are functions. Lambda calculus is appropriate for this: a value of type $(s_1 \times \ldots \times s_n \rightarrow s)$ can be written as

fun (x_1: s_1, …, x_n: s_n) t

where t is a term of sort s with free variables $x_1, \ldots x_n$. So the notation uses a variant of typed lambda calculus as in [CaW85]. – Assuming now that *person* is the name of a tuple type *tuple*(<(name, *string*), (age, *int*)>) and that *persons* is the name of a relation object of type *rel*(*person)*, we can formulate a query (“Find people older than 30”) in abstract syntax as:

select (persons, **fun** (p: person) >(age(p), 30)) whereas in concrete syntax it would be
persons **select**[age > 30] which is a simplification of a concrete version
persons **select**[**fun** (p: person) p age > 30]

This simplification can be recognized by the parser since with the application of **select** to *persons* the type of the parameter function can be determined and the full denotation of the function value be substituted for the abbreviation “age > 30”. However, in some cases, for example in optimization rules (see Section 5), the full notation for function values is needed and in fact very useful.

2.4 Programs

A parser/optimizer implementing the framework of this paper (let us call it an *SOS optimizer,* SOS for second-order signature) will accept *programs* of the *model level* and translate them through the use of *optimization rules* to the *representation level*. There is just a single language for programs (at both levels) and it is very simple, consisting of just five kinds of 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 statement assigns a value to an object; the value must match the type of the object. The fourth statement deletes an object, the fifth returns the value of an expression (a term built from *operators*) to the user or calling program. A little example program for the relational model defined before looks like this:

type city = *tuple*(< (name, *string*), (pop, *int*), (country, *string*) >)
type city_rel = *rel*(city)
create cities : city_rel
update cities := {enter values into the cities relation; dicussed later}
query cities **select**[pop > 500000]

It is interesting to note that through the use of denotable function values one can define (non-updatable) views without any special construct:

create french_cities : (→ city_rel)
update french_cities := **fun** () cities **select**[country = "France"]
query french_cities **select**[pop > 100000]

One can as well define parameterized views:

create cities_in : (*string* → city_rel)
update cities_in := **fun** (c: *string*) cities **select**[country = c]
query cities_in ("Germany")

3 Formal Concepts

In this section we give the formal “backbone” of the framework shown in Section 2, defining the concepts of *second-order signature (SOS)* and *second-order algebra (SOA)*. A second order signature is the semantics of a definition of a type system together with the specifications of polymorphic operators, as shown in Section 2. We do not yet formalize what specifications are and how they map into an SOS; that is left to future work. For the definitions we proceed in three steps: First, some well-known definitions of signature etc. are recalled. We then introduce extended signatures which

offer list sorts, product sorts, union sorts, and function sorts. Finally, SOS and SOA are defined.

3.1 Some Basic Definitions

Such definitions can be found, for example, in [EhGL89, EhM85].

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

(i) S is a set (whose elements are called *sorts*)

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

We call Σ an *S-sorted signature* and let Σ also denote the union of all sets $\Sigma_{w\,s}$. For $\omega \in \Sigma_{w\,s}$, $w = s_1 \ldots s_n$ denote the *argument sorts* and s the *result sort* and we often denote the operator as

$$\omega : s_1 \times \ldots \times s_n \to s \qquad \text{or as} \qquad s_1 \times \ldots \times s_n \;\; \to s \;\; \omega$$

Note that $n = 0$ is allowed and we call a 0-ary operator $\omega : \to s$ a *constant*. In some cases we want to distinguish between a subset of sorts T that can be used as argument sorts and a subset U whose elements can be used as result sorts. In that case we say Σ is a (T, U)-sorted signature. A signature has an associated set of *terms*:

Def. Given a signature (S, Σ), the set of *terms of sort s*, denoted *s-terms(*Σ), is defined as follows:

(i) A constant $\omega : \to s$ is a term of sort s.

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

Furthermore, *terms(*Σ) denotes set of of all terms over Σ.

Def. (*Terms with variables*) Let $X = \{X_s\}_{s \in S}$ be an S-indexed family of sets. We call $x \in X_s$ a *variable of sort s*. Given an S-sorted signature Σ, we denote by $\Sigma(X)$ the signature obtained by adding all variables in X as 0-ary operators. Then *s-terms*$(\Sigma(X))$ is the set of *terms with variables* of sort s.

An *algebra* defines the semantics of a signature. It consists of a set for each sort in the signature and a function on these sets for each operator in the signature:

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

(i) $S_A = \{s_A\}_{s \in S}$ where each s_A is a set (called the *carrier* of s).

(ii) $\Sigma_A = \{\sigma_A : s_{1,A} \times \ldots \times s_{n,A} \to s_A \mid \sigma \in \Sigma_{w\,s}, \text{ for } w = s_1 \ldots s_n\}_{w \in S^*,\, s \in S}$
where each σ_A is a function with the indicated domain and codomain sets in S_A.

3.2 Extended Signatures

The purpose of an extended signature is to introduce for a given set of sorts “automatically” list sorts, product sorts, union sorts, and function sorts, which can then be used to define operators. Examples have been shown in Section 2.

Def. Given a set (of sorts) S, an *extended S-sorted signature*, or *e-signature* for short, is a signature $(\bar{S}, \Sigma)$, where $\bar{S}$ is defined as follows:

(i) $s \in S \quad \Rightarrow \quad s \in \overline{S}$
(ii) 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 $\overline{S}$.
(iii) 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 $\overline{S}$.
(iv) If $s \in \overline{S}$, then s^+ is a sort in $\overline{S}$.
(v) If $s_1,\ldots, s_n$ and s are sorts in $\overline{S}$, for $n \geq 0$, then $(s_1 \times \ldots \times s_n \rightarrow s)$ is a sort in $\overline{S}$.

An extended signature has also an extended set of terms:

Def. For an S-sorted e-signature Σ the set of terms is defined as follows:

(i) If $\omega : \rightarrow s$ is a constant, then ω is a term of sort s. If $t_1, \ldots, t_n$ are terms of sorts $s_1,\ldots, s_n$, respectively, and $\omega : s_1 \times \ldots \times s_n \rightarrow s$ is an operator, then $\omega(t_1, \ldots, t_n)$ is a term of sort s.
(ii) If $t_1, \ldots, t_n$ are terms of sorts $s_1,\ldots, s_n$, respectively, then $(t_1, \ldots, t_n)$ is a term of sort $(s_1 \times \ldots \times s_n)$.
(iii) 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 $(s_1 \cup \ldots \cup s_n)$.
(iv) If $t_1, \ldots, t_n$ are terms of sort s, $n \geq 1$, then $<t_1, \ldots t_n>$ is a term of sort s^+.
(v) If $x_1, \ldots x_n$ are variables of sorts $s_1,\ldots, s_n$, for $n \geq 0$, and t is a term of sort s with free variables $x_1, \ldots x_n$, then

fun $(x_1: s_1, \ldots, x_n: s_n) \quad t$

is a term of sort $(s_1 \times \ldots \times s_n \rightarrow s)$. Furthermore, if $\omega : s_1 \times \ldots \times s_n \rightarrow s$ is an operator, then ω is a term of sort $(s_1 \times \ldots \times s_n \rightarrow s)$.

The definitions of terms with variables and of an algebra extend in the obvious way to extended signatures.

3.3 Second-Order Signatures

The underlying structure of our framework is that of a *second-order signature* which we define as follows:

Def. A *second-order signature (SOS)* is a quintuple $\Sigma = (K, \Gamma, T, \Delta, \Omega)$, where

(i) K is a set (whose elements are called *kinds*).
(ii) Γ is a $(K \cup T, K)$-sorted e-signature (the operators of Γ are called *type constructors*).
(iii) T is a set (whose elements are called *types*) defined as follows:
  (a) If $\gamma: \rightarrow k$ is a (constant) type constructor (i.e. $\gamma \in \Gamma_k$), then γ is in T.
  (b) If $\gamma: u_1, \ldots, u_n \rightarrow k$ is a type constructor, and for $t_1, \ldots, t_n$ holds:

$$\begin{cases} t_i \text{ is a type of kind } u_i \ (t_i \in u_i\text{-}terms(\Gamma)) & \text{if } u_i \in K \\ t_i \text{ is a term of type } u_i \ (t_i \in u_i\text{-}terms(\Omega)) & \text{if } u_i \in T \end{cases}$$

  then $\gamma(t_1, \ldots, t_n)$ is a type in T.

(iv) Δ is a $(K \cup T, K)$-sorted e-signature (elements are called *type operators*).
(v) Ω is a T-sorted e-signature (elements are called *operators*).

The purpose of this definition should be clear from the motivating examples in Section 2. There is a mutual dependency since types are formed using type constructors but type constructors are also

defined using types. A second-order signature is well defined, if there are no cycles in the dependency graph of definitions. To make sure that no cycles occur, one can require that an SOS is written as a sequence of type constructor, type operator, and operator definitions such that only concepts defined before are used in any definition. In Section 2 kinds and types were used together only in the definition of the *tuple* constructor. However, there are other important examples why it must be possible for type constructors not only to rely on kinds but also on types (or, put differently, not only on types, but also on values). Some of those are shown in later sections. A very simple example is the type constructor

int → DATA *string*

which allows to construct types such as *string*(4), *string*(20), … that is, fixed-length character strings. Here 4 and 20 are terms of type *int*. The issue is further discussed in Section 7. – One can assign semantics to a second-order signature by supplying a second-order algebra:

Def. Let $\Sigma = (K, \Gamma, T, \Delta, \Omega)$ be a second-order signature. A *second-order* Σ*-algebra (SOA)* is a triple $A = (T_A, \Delta_A, \Omega_A)$, where

(i) $T_A = \{t_A\}_{t \in T}$ is a family of sets (one set for each type in T)

(ii) Let $(K \cup T)_A = K_A \cup T_A$ with $K_A = \{k\text{-}terms(\Gamma)\}_{k \in K}$.

$((K \cup T)_A, \Delta_A)$ is a $(K \cup T, \Delta)$-algebra.

(iii) (T_A, Ω_A) is a (T, Ω)-algebra.

In other words, the semantics of a second-order signature is given by supplying a set for each type in T, a function for each type operator in Δ, and a function for each operator in Ω. Type operator functions in Δ_A operate on the carrier sets supplied for the types and the carrier sets for the kinds which are fixed to be the k-terms of Γ (the types of kind k formed by type constructors). Operator functions in Ω_A map between the carrier sets of the types. – In practice, a second-order algebra will be provided by implementation. To implement the model level (data model and query language) one needs only to realize the type mapping functions in Δ_A, since these are needed for parsing expressions. Optimization transforms programs at the model level to programs at the representation level. At that level one needs to provide data structures for the types (in fact, rather for the type constructors), procedures for the type operators, and procedures for the operators.

4 Modeling Representation and Query Processing

In Section 2 some examples of specification at the data model level have been shown; we now consider the level of representation and query processing. At the same time we motivate and introduce new specification techniques. A specification, regardless of the level, has the following structure of which not all parts need to be present:

kinds
type constructors
constructor specs
subtypes
operators

The new parts will be explained as we go along. The examples are drawn from relational query processing; we introduce streams and some storage structures with their access methods.

kinds IDENT, DATA, ORD, TUPLE, STREAM, SREL, TIDREL, BTREE, LSDTREE, RELREP

type constructors

	→ IDENT	*ident*
	→ DATA	*int*, *string*, *bool*, *point*, *rect*, *pgon*
	→ ORD	*int*, *string*
(*ident* × DATA)$^+$	→ TUPLE	*tuple*
TUPLE	→ STREAM	*stream*
TUPLE	→ SREL	*srel*
TUPLE	→ TIDREL	*tidrel*
TUPLE	→ RELREP	*relrep*

In this model data types that have an ordered domain (and therefore can be represented in a "one-dimensional" index structure) are distinguished in a kind ORD; there are also some data types for geometric objects. Tuples are defined as in Section 2. STREAM types represent streams of tuples in memory; it is assumed that the underlying execution engine can process sequences of operations on streams in a pipelined fashion. Sometimes streams need to be collected into temporary relations. These are represented by SREL types. A TIDREL type stands for a permanently stored relation with no specific order over which secondary index structures can be built. RELREP offers types that are generalizations of all relation representations. We now discuss two different ways of specifying B-trees (as a primary or "clustering" structure).

constructor specs

TUPLE × *ident* × ORD → BTREE *btree*

∀*tuple*: (*tuple*(*list*)) **in** TUPLE. ∀(*attrname*, *dtype*) **in** *list*.
btree(*tuple*, *attrname*, *dtype*)

This is a single-attribute B-tree. The *btree* constructor takes a tuple type, an identifier value, and a data type in ORD as arguments. The additional specification relates the three arguments and makes sure that the identifier and data type describe indeed one component of the tuple. This B-tree stores tuples according to one of their attribute values. An example B-tree type with this specification is *btree*(city, pop, *int*) where "city" denotes the city tuple type. Alternatively, one can more generally define B-trees organizing tuples by the value of some expression over the tuple with the following specification:

tuple: TUPLE × (*tuple* → *ord*: ORD) → BTREE *btree*

Here the first argument for the constructor is a tuple type, the second is a function value such that this function maps values of the particular tuple type selected as a first argument into values of some type in ORD. Here *btree*(city, pop) would be a valid type with the same meaning as in the first specification, but also

btree(city, **fun** (c: city) c pop **div** 1000)

The capability of indexing on derived values is not only interesting but even crucial for the LSD-tree [HeSW89], a structure capable of storing rectangles. It is used to store (tuples by) the bounding

boxes of polygons which are the values actually occurring as attributes (for example, geographical regions). The *lsdtree* constructor is defined as follows:

tuple: TUPLE × (*tuple* → *rect*) → LSDTREE *lsdtree*

If "state" is a tuple type of the form *tuple*(<(name, *string*), (region, *pgon*)>) and if there is an operator

pgon → *rect* **bbox** # (_)

then an LSD-tree organizing a "states" relation by bounding boxes of regions could be defined as

lsdtree(state, **fun** (s: state) **bbox**(s region))

A *subtype specification* introduces a partial order on types. This makes it possible to use subtype polymorphism in addition to the parametric polymorphism expressed by quantification over kinds. In some circumstances subtype polymorphism is easier to use. We employ it here to define a "generic" relation representation structure with a corresponding generic access operation.

subtypes

srel(*tuple*)	<	*relrep*(*tuple*)
tidrel(*tuple*)	<	*relrep*(*tuple*)
btree(*tuple*, *attrname*, *dtype*)	<	*relrep*(*tuple*)
lsdtree(*tuple*, *f*)	<	*relrep*(*tuple*)

Intuitively, this says, for example, that any BTREE type of the form *btree*(*tuple*, *attrname*, *dtype*) may be viewed as a *relrep*(*tuple*) type. That is, any operator applicable to the latter is applicable to the former. Obviously, each line of this specification establishes a partial order between infinitely many pairs of types. Argument variables on the right side must also appear on the left side (hence we have generalization from left to right). We can now use this subtype relationship to define an operator "feeding" tuples from any relation representation into a stream:

operators

∀*relrep*: *relrep*(*tuple*) **in** RELREP. *relrep* → *stream*(*tuple*) **feed** _ #

Next we specify some search operations on index structures. A range query operator on a B-tree (using the first type definition above) is defined as follows:

∀*btree*: *btree*(*tuple*, *attrname*, *dtype*) **in** BTREE.
btree × *dtype* × *dtype* → *stream*(*tuple*) **range** _ _ _ #

The operator also supports queries with halfranges if values like – ∞ and + ∞ are available. These can be provided by a specification:

∀*ord* **in** ORD. → *ord* **bottom**, **top** #

For the LSD-tree, storing rectangles, we provide two search methods. The first retrieves all rectangles containing a query point, the second all rectangles overlapping a query rectangle. The tuples whose rectangles qualify are fed into a stream.

∀*lsdtree*: *lsdtree*(*tuple*, *f*) **in** LSDTREE.
lsdtree × *point* → *stream*(*tuple*) **point_search** _ _ #
lsdtree × *rect* → *stream*(*tuple*) **overlap_search** _ _ #

It is also possible, although a little more involved, to specify multi-attribute structures (see [Lo92]) indexing by an arbitrary number of dimensions together with operators for the well-known query types such as multi-dimensional exact-match, partial-match, or range queries. Or one can define a multi-attribute B-tree (ordered first by one attribute, then for equal values by a second attribute, etc) together with a query operator specifying values for a prefix of the attributes used for indexing. For lack of space such definitions are not shown here.

A substantial part of query processing is provided by stream operators. Here are some:

$\forall stream$: $\underline{stream}(tuple)$ **in** STREAM.
$stream \times (tuple \rightarrow \underline{bool}) \quad \rightarrow stream \quad$ **filter** _ #[_]

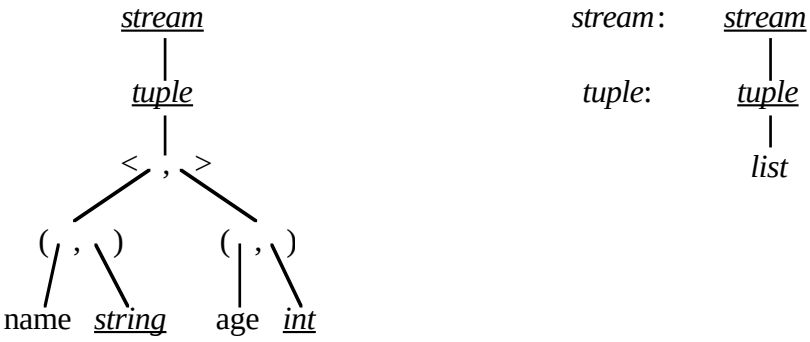
This operator filters tuples in the stream by a selection condition.

$\forall stream$: $\underline{stream}(tuple)$ **in** STREAM. $\forall data_i$ **in** DATA.
$stream \times (\underline{ident} \times (tuple \rightarrow data_i))^+ \quad \rightarrow s$: STREAM **project** _ #[_]

This is a generalized projection on stream tuples. The second argument is a list of pairs. In each pair the first component is a possibly new attribute name for the result tuple, the second component is a function to compute the value of this attribute (which can also be given simply as an old attribute name). Note that the type of the result stream is determined by the operator itself. This operator realizes at the representation level operators like λ or *extend* in the NST-algebra [GüZC89] or *replace* in the algebra of [AbB88]. – For convenience we add a simpler operator useful for udates which replaces an attribute value in a tuple by a new one:

$\forall stream$: $\underline{stream}(tuple$: $\underline{tuple}(list))$ **in** STREAM. $\forall(attrname, dtype)$ **in** *list*.
$stream \times attrname \times (tuple \rightarrow dtype) \quad \rightarrow stream \quad$ **replace** _ #[_ , _]

This example illustrates that a *pattern* in an operator specification is really a term tree where some subtrees may have been cut off and replaced by variables, and where internal nodes may also be labeled by variables (Figure 1). Note that this feature of associating variables with the nodes of a term tree (describing a type) allows for a very precise specification of polymorphic operations.



(a) a term tree (b) a pattern matching this term tree

Figure 1

An operator may be useful that collects a stream into a temporary relation:

$\forall stream$: $\underline{stream}(tuple)$ **in** STREAM. $\quad stream \rightarrow \underline{srel}(tuple) \quad$ **collect** _ #

As a final example, we show the specification of a nested loop join operator. Recall that this join method combines tuples of an “outer” relation, which usually comes as a stream, with tuples of an

"inner" relation. The term "nested loop" seems to imply that the inner relation is scanned to find matching tuples for each tuple of the outer one, but this is not necessarily true: It is also possible that the inner relation is accessed through an index. Hence "nested loop" is misleading; we call it a "search join".

This is not easy to describe algebraically. One would like to have a general search join operator that is able to work with scanning or any kind of index access on the inner relation. One difficulty is that a value from the current outer tuple must be fed into a search condition or method to access the inner relation. In [BeG92] we did not succeed; it was necessary to introduce one join operator for each kind of index access available. The more powerful framework of this paper provides an elegant solution:

$\forall stream_1$: *stream*($tuple_1$) in STREAM. $\forall stream_2$ in STREAM.
$stream_1 \times (tuple_1 \rightarrow stream_2) \rightarrow s$: STREAM **search_join** _ _ #

The first argument is a stream (for the outer relation), the second a function that yields for each tuple of the outer relation a stream of (matching) tuples of the inner relation. We illustrate the use of this operator in the following examples of query processing (that is, programs at the representation level).

```
type city          = tuple(<(cname, string), (center, point), (pop, int)>)
type state         = tuple(<(sname, string), (region, pgon)>)
create cities_rep  : btree(city, pop, int)
create states_rep  : lsdtree(state, fun (s: state) bbox(s region))
   {fill cities_rep and states_rep with values}
```

We now form a join of the cities with the states they lie in. At the model level this would be a query (with an **inside** operator defined in the obvious way):

```
query  cities  states  join[center  inside  region]
```

At the representation level, we implement this through the search join, first by repeatedly scanning the states relation in its LSD-tree representation.

```
query
   cities_rep  feed
   fun (c: city)  states_rep  feed  filter[fun (s: state)  c center  inside  s region]
   search_join
```

You may check that the second argument is indeed a function returning for each city tuple a stream of state tuples (consisting in fact of the one state tuple containing the city). The search join operator itself takes each tuple of its first argument stream and applies to it the second argument function which returns a stream. Each tuple in this stream is combined with the given tuple of the first stream and pushed into the output stream of the search join. We now use the same search join operator to implement repeated search on the LSD-tree representing the states relation:

```
query
   cities_rep  feed
   fun (c: city)  states_rep  (c center)  point_search
      filter[fun (s: state)  c center  inside  s region]
   search_join
```

Note that the filtering of the stream by condition “c center **inside** s region” is necessary because the point-search returns all tuples where the bounding box of the state's region contains the city point.

5 Optimization Rules

In this section we briefly sketch how optimization rules can be formulated in the framework of this paper. As an example we show a rule to translate a join by a geometric “inside” condition into a repeated search on an LSD-tree representation, corresponding to the last example of the previous section:

∀rel_1: $\underline{rel}(tuple_1)$ **in** REL. ∀rel_2: $\underline{rel}(tuple_2)$ **in** REL. ∀*point*: ($tuple_1 \rightarrow \underline{point}$).
∀*region*: ($tuple_2 \rightarrow \underline{pgon}$).

rel_1 rel_2 **join**[**fun** (t_1: $tuple_1$, t_2: $tuple_2$) (t_1 *point*) **inside** (t_2 *region*)]
→ rep_1 **feed**
fun (t_1: $tuple_1$) lsd_2 (t_1 *point*) **point_search**
filter[**fun** (t_1: $tuple_1$, t_2: $tuple_2$) (t_1 *point*) **inside** (t_2 *region*)]
search_join

if rep(rel_1, rep_1) **and** rep_1: $\underline{relrep}(tuple_1)$
and rep(rel_2, lsd_2) **and** lsd_2: $\underline{lsdtree}(tuple_2, f: tuple_2 \rightarrow \underline{rect})$.

The rule first declares some variables and then states that the pattern before the arrow can be translated into the pattern following the arrow if certain conditions hold. The conditions are that rel_1 must be represented by some object rep_1 which must have type $\underline{relrep}(tuple_1)$ and a similar condition for rel_2 and lsd_2. Obviously, testing the predicate “rep(rel_1, rep_1)” must be a catalog lookup; what a catalog is and how it is filled is explained in the next section.

This is an adaptation of the rules in [BeG92] used in the Gral system; see that paper for an explanation of our overall approach to rule-based optimization which may be extended to build an SOS optimizer. Whereas there various kinds of variables to represent relations, attribute names, parameter expressions, etc. had to be introduced in an ad-hoc manner, such variables and also type-checking predicates follow naturally from the framework of this paper. Of course, there remains a lot of work to be done to extend the approach of [BeG92], which dealt only with relational optimization of queries, to more general data and representation models and to updates.

6 Updates

It is now necessary to refine the notions of model level and representation level and the role of optimization. We have stated before that an SOS optimizer will accept programs of the model level and translate them into programs of the representation level, which may be executed directly. Ideally this should be the case. However, it implies that the system determines on its own the appropriate representation objects for objects at the model level. For example, the system would need to determine which relation representation structure should be used and which indices need to be created. With current technology for most data models and representation models this is not realistic. Instead, a database administrator will usually decide on the representation.

To take that into account, we drop the strict separation between model level and representation level and consider “mixed” type systems with model as well as representation types and operators. Another reason why this is useful is the fact that often some types occur at both levels, for example, atomic data types, or a tuple type. However, the system must be able to distinguish between model, representation, and hybrid types and operators, because model types and operations need to be translated or optimized whereas the other can be executed directly. One can distinguish, for example, by introducing sections

model type constructors
rep type constructors
hybrid type constructors

in a specification. Operators can be recognized by their result type. A database system with an SOS optimizer front-end will process “mixed” programs as follows:

- Type definition statements are processed internally by the optimizer.
- Object creation and deletion for model types leads to catalog management operations. For representation or hybrid types corresponding DBMS procedures are also called.
- Updates and queries whose result type is a model type must use only model and hybrid types and operations. The optimizer transforms these (through optimization rules) into equivalent updates and queries that use only hybrid or representation types and operations. These are given to the DBMS for execution.
- Updates and queries whose result is a hybrid or representation type must use only hybrid or representation types and operations. These are passed directly to the DBMS.

To accomodate updates in the algebraic framework we view updates as functions modifying one of their arguments. More specifically, we introduce a special class of functions called *update functions*, characterized as follows:

(i) They fulfill a special condition: The type of first argument and result must be the same.
(ii) Their application has a special effect: The result of the function application is assigned to the first argument. Hence, the first argument must be an object (a “variable”), not an expression.

We denote update functions by a special symbol “$\bullet\!\!\rightarrow$”. Update functions can be described by operator specifications like all other operations. – For illustration we design update operators for the relational model and for B-trees as one possible representation. Prefix syntax is used for all update operators.

$\forall rel$: $\underline{rel}(tuple)$ **in** REL.

	$\rightarrow$	*rel*	**empty**
$rel \times tuple$	$\bullet\!\!\rightarrow$	*rel*	**insert**
$rel \times rel$	$\bullet\!\!\rightarrow$	*rel*	**rel_insert**
$rel \times (tuple \rightarrow \underline{bool})$	$\bullet\!\!\rightarrow$	*rel*	**delete**

$\forall rel$: $\underline{rel}(tuple$: $\underline{tuple}(list))$ **in** REL. $\forall (attrname, dtype)$ **in** *list*.

$rel \times (tuple \rightarrow \underline{bool}) \times attrname \times (tuple \rightarrow dtype)$ $\bullet\!\!\rightarrow rel$ **modify**

The meaning of these operators should be clear. For example, the “modify” operator changes all tuples qualified by the second argument predicate by applying the fourth argument function to the tuple and assigning this value to attribute *attrname*. – At the representation level, the first few update operators for B-trees correspond directly to the relational counterparts:

$\forall btree$: $\underline{btree}(tuple, attrname, dtype)$ **in** BTREE.

	$\rightarrow$	*btree*	**empty**
$btree \times tuple$	$\bullet\!\rightarrow$	*btree*	**insert**
$btree \times \underline{stream}(tuple)$	$\bullet\!\rightarrow$	*btree*	**stream_insert**

For the design of deletion and tuple update operations we observe that each of these tasks consists of two parts. The first is to find the tuples concerned and the second to either delete them or to change their value. In order to achieve an orthogonal design, that is, to avoid a proliferation of update operators, the search method should be selectable independently from the update or deletion operator. Search methods are, for example, scanning the B-tree leaves, searching through the index part, or accessing tuples through a sequence of tuple identifiers delivered from a secondary index. This leads to the following design:

$btree \times \underline{stream}(tuple)$	$\bullet\!\rightarrow$	*btree*	**delete**

Here the meaning is that each tuple in the second argument stream is present in the B-tree; the operator deletes it from the structure. The stream will normally be created by a search operation on the same B-tree. We assume that the stream mechanism connects the search and the deletion operation in such a way, that the position of a tuple in the structure determined by search is still available when the tuple "returns" from the stream for deletion. For example, the tuple is still fixed on a buffer page. – For tuple update two different operators are needed:

$btree \times \underline{stream}(tuple) \times (\underline{stream}(tuple) \rightarrow \underline{stream}(tuple))$

$\bullet\!\rightarrow$	*btree*	**modify**, **re_insert**

The second argument determines again which tuples are concerned. The third argument is a function modifying the tuples by stream operators (e.g. "replace"). Although the parameters are the same, the two operators behave differently. The first modifies tuples in situ. The second is to be used for key updates; the operator deletes the tuple from the current position and reinserts it according to the new key value.

In a final example, we illustrate the use of update operators as well as the "processing behaviour" of an SOS optimizer. We also explain the notion of "catalog". We assume for the time being that *catalog* is a predefined type that essentially describes *n*-ary relations whose components may in particular be names of objects created in a database. Hence the type constructor might be defined as:

$(\text{IDENT} \cup \text{DATA})^+$	$\rightarrow \text{CATALOG}$	*catalog*

The only special thing about this type is that tests whether tuples are present can be written like PROLOG predicates within an optimization rule (see Section 5). Catalog objects are used to establish connections between a model object and its representation objects. In the following example, we assume a catalog "rep" has been created together with the database:

R **create** rep : *catalog*(*ident*, *ident*)

Letters M, R, and H indicate, whether a statement belongs to the model or representation level or is hybrid. A leading "⇒" indicates that this statement was created by the optimizer. The following might be a sequence of statements processed by an SOS optimizer:

H **type** city = *tuple*(<(cname, *string*), (center, *point*), (pop, *int*)>)

M **create** cities : *rel*(city)
R **create** cities_rep : *btree*(city, pop, *int*)
R **update** rep := **insert** (rep, cities, cities_rep)

Here the connection between the "cities" relation and the representing B-tree "cities_rep" is entered into the catalog (by a special **insert** operation defined for *catalog* types).

H **create** c : city
H **update** c := {create a city tuple at the user interface}
M **update** cities := **insert** (cities, c)
⇒ R **update** cities_rep := **insert** (cities_rep, c)
{more tuples are inserted}
M **update** cities := **delete** (cities, pop ≤ 10000)
⇒ R **update** cities_rep := **delete** (cities, cities bottom 10000 **range**)

So here the tuples to be deleted are found by a range search on the B-tree. Finally, update of the key attribute might be translated (finding tuples this time by scanning the B-tree leaves):

M **update** cities := **modify** (cities, country = "India", pop, pop * 1.1)
⇒ R **update** cities_rep :=
re_insert (cities_rep, cities_rep **feed** **filter**[country = "India"],
fun (s: *stream*(city)) s **replace**[pop, **fun** (c: city) (c pop) * 1.1]))

7 Related Work and Discussion

Second-order signature, as we have defined it, 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 (polymorphic) typed lambda calculus invented independently by Girard [Gi72] and Reynolds [Re74]. Milner [Mil78] showed that in many cases the types of polymorphic functions can be inferred without explicitly mentioning type parameters. This led to ML and other functional programming languages. Cardelli and Wegner [CaW85] give a survey of types and polymorphism in programming languages; a survey of the underlying theory is presented in [Mit90].

A major aspect of recent research in type systems in programming languages as well as databases has been the modeling of subtypes and inheritance known from object-oriented programming (e.g. [Ca84, BrW86]). In [CaW85] it is also shown how parametric polymorphism viewed as *universal quantification* over all types can be integrated with subtype polymorphism [Ca84] viewed as *bounded quantification* over all subtypes of a given type.

Closer yet to our work is the proposal of a three-level type structure [Ca88, CaL90] consisting of (i) values including functional values, (ii) types and type operators, and (iii) kinds, where types classify values and functions, and kinds classify types and type operators. The notion of kinds as "the types of types" is taken from [McC79]. Cardelli [Ca88] describes various type systems with e.g. a single type, many types, one kind, and many kinds. TYPE is introduced as the kind containing all types and POWER[A] for a type A as the kind containing all subtypes of A. Hence universal and bounded quantification from [CaW85] can now both be viewed as quantification over kinds. The three-level

structure is further pursued in the design of a programming language *Quest* [Ca89] for which [CaL90] gives formal typing rules and semantics. Since the three-level structure occurs also in SOS, we discuss similarities and differences in more detail below.

In database systems, one standard approach to offer querying facilities for a new data model is to define a (query) algebra. Although an algebra is nicely summarized by its signature, it is interesting to note that one rarely sees signatures for query algebras. This is due to the fact that database algebras need a level of polymorphism that cannot be expressed by simple signatures. Therefore the types of operands and results are usually described by some ad-hoc formalism. The difficulty of describing operations like that of relational algebra within type systems has been observed particularly by researchers in database programming languages (e.g. [Stem90]). Here a main goal is to make static type checking possible for query expressions. Perhaps the most successful attempt so far is the language Machiavelli [OhBB89] where, for example, projection and join operations can be defined and the result types of operations be inferred. A key idea to achieve this is a generalization of the relational model by a partial order on tuple (or record) types [Oh88, OhB88]. Recently, the notion of kinds has also been used in [BuO91] to model subtyping; a kind describes all record types which have a given set of components (labels) – this is related to the POWER[A] kind of [Ca88].

There is also some work related to SOS on the algebraic specification side. In [LeW91] “higher order signatures” are mentioned and it is observed that there exist *sort constructors* such as *seq*: Sort → Sort which for a given specification form a (single-sorted) signature. However, in detail their approach to higher-order signatures is quite different from ours and more complex; for example, there are also sorts representing whole signatures and specifications.

A major distinction between SOS and programming language type systems is that SOS has an “external semantic component” which is reflected in the separation of second-order signature and second-order algebra. This results from the purpose of SOS which is to represent a data model and query language (or a representation and query processing system) at such a level of abstraction that parsing (type checking) is possible and that optimization rules can be described concisely. An SOS parser/optimizer essentially works only with the second-order signature. The second-order algebra $(T_A,\ \Delta_A, \Omega_A)$ is needed for the system to work properly but only as “black boxes”: T_A gives a meaning to the types (carrier sets in the data model and data structures at the representation level) but the optimizer does not need to know this. Δ_A describes the type mapping for complex operators like project, join; the parser just calls the corresponding procedure (after recognizing the operator) but does not need to know how it works. Finally, Ω_A gives polymorphic functions for the operators. At the data model level these functions are not really needed since applications of the operator are translated by optimization to operators of the representation level. At the representation level these functions correspond to procedures available in the system; they are called to evaluate a query plan but this is not interesting any more to parsing and optimization.

These general remarks apply in particular when we compare SOS to Cardelli's three-level approach [Ca88, CaL90]. Main differences are:

- There is no concept of separating signature and algebra. Whenever entities are defined (e.g. types, polymorphic functions, type operators) their semantics are given directly, forming concepts “bottom-up”, as usual in programming languages.
- The idea of SOS to view *types as terms* (that are in a sense uninterpreted) is not present here. So there is no distinction between *type constructors* and *type operators*. In [Ca88] type

operators always have an associated type function and so correspond to our type operators, not our type constructors.
- In connection with the "types are terms" concept, the idea of choosing a collection of kinds arbitrarily to define functionalities of type constructors, is also absent.

Two other important features of SOS should be discussed. The first is the possibility to define result types of operations (such as *rel*: REL for **join** meaning "some type *rel* in kind REL") externally (remember that such a specification maps to the use of a type operator). An alternative is to try to describe the result type precisely within the specification, for example, by writing the result type as a term over the operand types with suitable type operators (e.g. tuple concatenation for join). Our reasons for offering this possibility are two-fold: (1) We feel that a database algebra operation should be viewed as consisting in general of two parts, namely, a schema mapping and an instance mapping. Therefore the type mapping is an essential part of the operation and should be described explicitly. (2) Some operations (e.g. *nest* in complex object algebras) have a too complex type mapping to be described concisely in a specification. It must be possible to define it separately.

The other important feature is the fact that types are defined over other types *as well as values*. This makes the definition of SOS a bit more complex, but it is quite natural and also crucial for the definition of some types (see the B-tree and LSD-tree examples of Section 4). In [Ca88] another solution is suggested: If one wants to define a type function as a conditional ("*if B then* $type_1$ *else* $type_2$) then the function depends on three arguments the first of which (in our view) is a *value* of *type* Bool whereas the other two are *types* of *kind* TYPE. Cardelli suggests to "lift" Bool to the kind level, so that BOOL is a kind with two types True and False. While this technical trick helps to keep the value and type levels separate, we find our solution to include values into type definitions much more natural.

Clearly, the work described in this paper is only a first step. We have defined SOS as the formal core of a specification framework and shown applications to relational modeling, query processing, updates, and optimization. Future work might include the following:
- Try to define more complex data models and algebras for queries or query processing within the SOS framework, such as, for example, [AbH84, ShZ89, ScS90, VaD91].
- This study should clarify how much can be done with the specification techniques described and which additional techniques are needed.
- Define the semantics of specifications by mapping them into a second-order signature.
- Implement an SOS parser/optimizer.

Acknowledgments

Initial ideas about "second-order" algebra were developed jointly with Martin Erwig. The work described in this paper was done while the author was a guest of Hans-Jörg Schek at the Institut für Informationssysteme, ETH Zürich, during a sabbatical stay. I thank Hans-Jörg for his support and friendship. Gisbert Droege, Gerhard Weikum, Hans-Jörg Schek, Martin Erwig, and Ludger Becker have contributed through interesting discussions about SOS and by helpful comments on a draft version of this paper. Martin Wunderli, Helmut Kaufmann, and Andreas Wolf also helped to clarify concepts. I thank the whole group at ETH, in particular Antoinette Förster, for making my stay in Zürich very pleasant.

References

[AbB88] Abiteboul, S., and C. Beeri, On the Power of Languages for the Manipulation of Complex Objects. Technical Report 846, INRIA (Paris), 1988.

[AbH84] Abiteboul, S., and R. Hull, IFO: A Formal Semantic Database Model. Proc. ACM Conf. on Principles of Database Systems, 1984, 119-132.

[BaK86] Bancilhon, F., and S. Khoshafian, A Calculus for Complex Objects. Proc. ACM Symposium on Principles of Database Systems (Cambridge, Mass.), 1986, 53-59.

[Bato88] Batory, D.S., J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise, GENESIS: An Extensible Database Management System. *IEEE Trans. on Software Engineering 14 (1988)*, 1711-1730.

[BeG92] 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.

[BuO91] 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.

[BrW86] Bruce, K.B., and P. Wegner, An Algebraic Model of Subtypes in Object-Oriented Languages. *SIGPLAN Notices 21 (1986)*, 163-172.

[Ca84] Cardelli, L., A Semantics of Multiple Inheritance. Symposium on the Semantics of Data Types (LNCS 173), 1984, 131-144.

[Ca88] Cardelli, L., Types for Data-Oriented Languages. Proc. Int. Conf. on Extending Data Base Technology (LNCS 303), 1988, 1-15.

[Ca89] Cardelli, L., Typeful Programming. SRC Report 45, Digital Equipment Corporation, Palo Alto, CA, 1989.

[CaL90] Cardelli, L., and G. Longo, A Semantic Basis for Quest. Proc. ACM Conf. on LISP and Functional Programming, 1990, 30-43.

[CaW85] Cardelli, L., and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys 17 (1985)*, 471-522.

[CrMW87] Cruz, I.F., A.O. Mendelzon, and P.T. Wood, A Graphical Query Language Supporting Recursion. Proc. ACM SIGMOD 1987, 323-330.

[EhGL89] Ehrich, H.D., M. Gogolla, and U.W. Lipeck, Algebraische Spezifikation abstrakter Datentypen. Teubner, Stuttgart, 1989.

[EhM85] Ehrig, H., and B. Mahr, Fundamentals of Algebraic Specification I. Springer, Berlin, 1985.

[ErG91] Erwig, M., and R.H. Güting, Explicit Graphs in a Functional Model for Spatial Databases. FernUniversität Hagen, Informatik-Report 110, 1991.

[Gi72] Girard, J.Y., Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur. Ph.D. Thesis, Université Paris VII, 1972.

[GrD87] Graefe, G., and D.J. DeWitt, The EXODUS Optimizer Generator. Proc. ACM SIGMOD 1987, 160-172.

[Gü89] Güting, R.H., Gral: An Extensible Relational Database System for Geometric Applications. Proc. of the 15th Intl. Conf. on Very Large Data Bases, 1989, 33-44.

[GüZC89] Güting, R.H., R. Zicari, and D.M. Choy, An Algebra for Structured Office Documents. *ACM Transactions on Information Systems 7 (1989)*, 123-157.

[GyPV90] Gyssens, M., J. Paredaens, and D. van Gucht, A Graph-Oriented Object Database Model. Proc. ACM Conf. on Principles of Database Systems 1990, 417-424.

[Haas89] Haas, L.M., J.C. Freytag, G.M. Lohman, and H. Pirahesh, Extensible Query Processing in Starburst. Proc. ACM SIGMOD 1989, 377-388.

[HeSW89] Henrich, A., H.-W. Six, and P. Widmayer, The LSD Tree: Spatial Access to Multidimensional Point- and Non-Point-Objects. Proc. Intl. Conf. on Very Large Data Bases 1989, 45-53.

[LeW91] Leszczylowski, J., and M. Wirsing, Polymorphism, Parameterization, and Typing: An Algebraic Specification Perspective. Proc. Symposium on Theoretical Aspects of Computer Science (LNCS 480), 1991, 1-15.

[Lo92] Lomet, D., A Review of Recent Work on Multi-attribute Access Methods. *ACM SIGMOD Record 21 (1992)*, 56-63.

[[McC79] McCracken, N., An Investigation of a Programming Language with a Polymorphic Type Structure. Ph.D. Thesis, Syracuse University, 1979.

[Mil78] Milner, R., A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences 17 (1978)*, 348-375.

[Mit90] Mitchell, J.C., Type Systems for Programming Languages. In: J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, 1990, 365-458.

[Oh88] Ohori, A., Semantics of Types for Database Objects. Proc. Int. Conf. on Database Theory 1988, 239-251.

[OhB88] Ohori, A., and P. Buneman, Type Inference in a Database Programming Language. Proc. ACM Conf. on LISP and Functional Programming, 1988.

[OhBB89] 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.

[Re74] Reynolds, J.C., Towards a Theory of Type Structure. Programming Symposium, Paris (LNCS 19), 1974, 408-425.

[Sche90] 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.

[ScS91] Schek, H.J., and M.H. Scholl, From Relations and Nested Relations to Object Models. In: M.S. Jackson and A.E. Robinson (eds.), Aspects of Databases. Proc. of the 9th British National Conf. on Databases, Wolverhampton 1991, 202-225.

[ScS90] Scholl, M.H., and H.J. Schek, A Relational Object Model. Proc. Int. Conf. on Database Theory, Paris, France, 1990, 89-105.

[ShZ89] Shaw, G.M., and S.B. Zdonik, An Object-Oriented Query Algebra. Proc. 2nd Int. Workshop on Database Programming Languages, 1989, 103-112.

[Stem90] Stemple, D., L. Fegaras, T. Sheard, and A. Socorro, Exceeding the Limits of Polymorphism in Database Programming Languages. Proc. Int. Conf. on Extending Data Base Technology (LNCS 416), 1990, 269-285.

[StR86] Stonebraker, M., and L.A. Rowe, The Design of POSTGRES. Proc. ACM SIGMOD 1986, 340-355.

[VaD91] Vandenberg, S.L., and D.J. DeWitt, Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance. Proc. ACM SIGMOD 1991, 158-167.