EAV: An Algebra for the Representation and Realization of Internal, Conceptual Objects in Configurable Database Management Systems

Andreas Geppert, Klaus R. Dittrich

Technical Report 93.25

Institut für Informatik, Universität Zürich
Winterthurerstr. 190
CH-8057 Zürich, Switzerland
{geppert, dittrich}@ifi.unizh.ch

July 31, 1992
Revised Version May 24, 1993

Abstract

The aim of the KIDS[1] project is the specification and generation/configuration of database management systems (DBMS). Thus, as for one aspect of KIDS, the modelling of potentially arbitrarily structured (generic) objects on different levels of abstraction has to be supported. First, a meta model provides for the description of data and operations close to the view of the user. Second, data are represented as *physical objects* on the storage level. Hence, a mapping from data model objects into physical objects is desired. Since that mapping is likely to get very complex, an intermediate representation is introduced, and the mapping is subdivided into two steps. On the intermediate level, data is represented in the form of *triples* (entity, attribute, values) or as quadruples (which allow the ordering of specific sets of triples). Triples are comparable to logical objects in database management systems: neither are they stored physically, nor are they visible to users. The description of data model objects results in a *triple scheme*, while the semantics of data model operations can be expressed in terms of operations on triple sets. Furthermore, the way sets of triples have to be stored is specified by means of *reduction rules*. The *EAV*[2] algebra supports the specification of triples and quadruples, operations on sets of triples, and the definition of (reduction) rules by use of algebraic expressions.

1. ***K***ernel Machine-Based ***I***mplementation of ***D***atabase Management ***S***ystems
2. ***E***ntity-***A***ttribute-***V***alue

Contents

1 Introduction

KIDS [Dittrich 91] supports the configuration/generation of (parts of) database management systems in a declarative manner. Starting from an abstract representation of data model objects[3], specific aspects of a database system (e.g., integrity maintenance and transaction management in addition to a data model) are modeled by means of rules. Those rules in turn serve as input for generator and configuration tools, which generate or configure DBMS components providing for the functionality specified by the rules.

One major part in the KIDS environment is the specification of data model objects (e.g., relations and tuples, or objects in the sense of object-orientation) and the mapping of those objects to physical objects (*reduction rules*). While a fixed notion of physical objects can be assumed, the structure and behaviour of data model objects may vary corresponding to the respective data model. Hence, the mapping of data model objects to physical objects has to be flexible in order to support various data models adequately.

3. Throughout this paper, we use the term “object” in a generic way, i.e. not as the representation of concrete real-world entities. Furthermore, “object” is used as a synonym of data item, and is not meant in the sense of object-orientation.

In this paper, we elaborate on (parts of) the mapping of data model objects and operations into physical ones. Our belief is that a *one-step* mapping is not adequate for data model properties. In the case of a one-step transformation, the physical representation had to be specified for each kind of "object" as well as the effect of data manipulation operations on physical objects. Not only would that mapping be too complex, it would also imply a rather bad modularized structure of the object management subsystem. Additionally, the inclusion of functionality like logical access paths, optimization, distribution and the like could not be supported well.

Therefore, we propose two mappings, one from data model objects into logical ones, and one from the latter into physical objects. As usual in DBMS, logical objects are just an intermediate representation, but do not exist physically. In addition, these objects are termed "conceptual" since their representation is simply a realization aid during DBMS construction. Essentially, after a data model has been specified (in terms of logical objects), code will be generated for the realization of logical objects. Hence, the combination of the mapping of data model concepts into logical objects and the generated code results in a direct mapping from the data model level to the physical level.

Thus, since logical objects do not exist physically, they need not be efficient neither in terms of space nor in terms of (run) time. Rather is it a crucial issue to support a flexible, powerful, and concise framework for specification.

The framework for the specification (and realization) of logical objects is described in this paper. Our approach is based on an algebra (based on the experiences made with algebraic specification, with algebras for the formalization of data models, and optimization of query languages in DBMS), and possesses similarities to conceptual models like the ER model [Chen 76] or the Telos model [Mylopoulos 90].

The reminder of the paper deals with the *EAV* model. In the next section, an overview on the data model aspect of KIDS is given. In section 3, we given an informal introduction to the underlying EAV algebra (a formal definition can be found in the appendix), while section 4 covers reduction of logical objects.

2 Overview

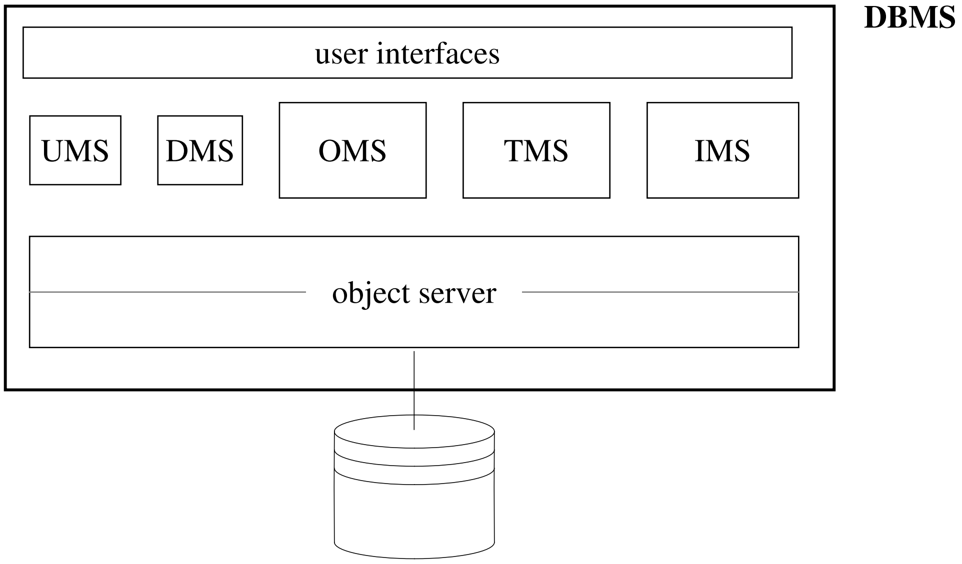
The aim of the KIDS project is to develop an environment that supports the generation and configuration of various DBMS components. In order to provide for a high degree of modularity and independence between the tasks of a DBMS, a subsystem architecture is proposed [Geppert 92a] (see Figure 1). In this paper, we concentrate on the data model aspect, which is realized by the object management subsystem on top of the KIDS object server [Nittel 92].

Since the task to generate an object management subsystem is still quite complex, a two-step mapping is applied that distinguishes between data model, logical, and physical objects. The corresponding component architecture is given in Figure 2. The first mapping specifies data model structures and operations in terms of logical objects and attached operations. The relationship (i.e., the mapping) between data model objects and logical objects has to be given by the DBI, i.e., it is she who realizes data model concepts. Logical objects, on the other hand, are mapped to physical objects (*byte containers).* This mapping, in turn, is automatic, i.e., based on the logical object specification a tool will generate software that implements the specification given by the DBI. This procedure is depicted in Figure 3.

Note that this mappings and generations occur at specification time. At runtime, when the (generated) DBMS executes, logical objects will not exist, since they are solely a means to bridge the gap between data model and physical objects conceptually.

While the requirements for a data model specification language and the language itself are described elsewhere [Geppert 92b, Scherrer 93], this paper elaborates on the specification and realization of logical objects.

Currently, both, types as well as instances (of logical objects) are specified in terms of triples. A triple is a tuple consisting of an entity identifier, an attribute name, and an attribute value. The value in turn may be the identifier of a triple or an attribute name (of other tri-



OMS = object management, TMS = transaction management, IMS = integrity maintenance, UMS = user and group management, DMS = logical database manager

OMS = object management, TMS = transaction management, IMS = integrity maintenance, UMS = user and group management, DMS = logical database manager, SMS = schema manager, CM = communication manager, EM = event manager

Figure 1 Generated/Configured DBMS Subsystem Architecture

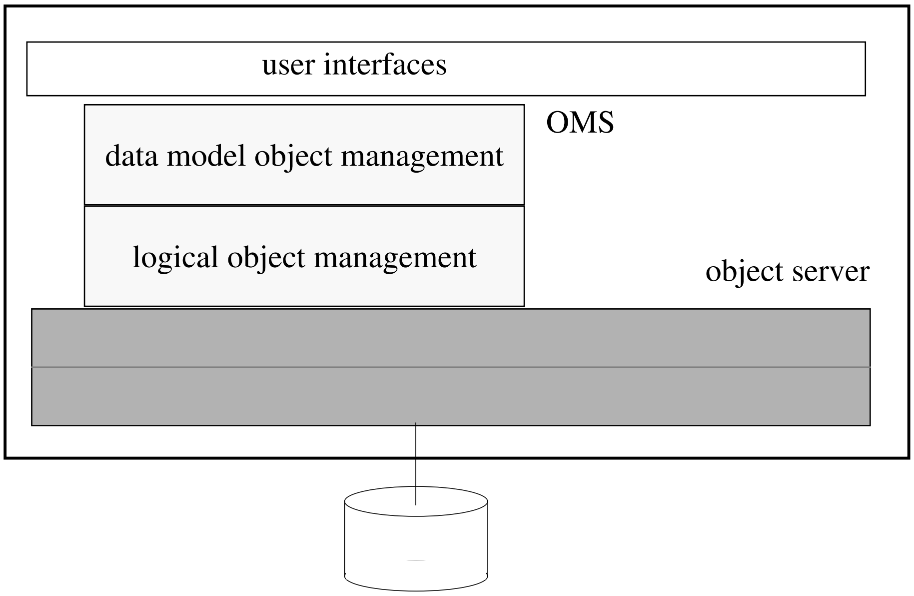


Figure 2 Component Architecture of Object Management Subsystem

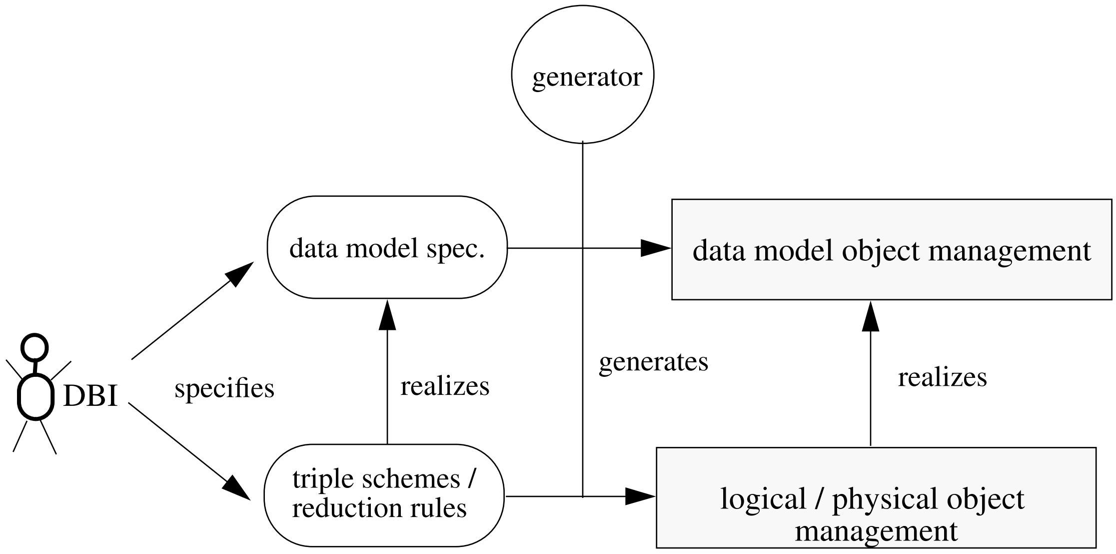


Figure 3 Specification / Generation Process

ples), such that links between triples can be easily represented. Quadruples possess an additional fourth component (an index) which supports ordering of triples with equal entity identifiers and attribute names, respectively. For a specific (meta) type like *relation*, *triple schemes* are specified that determine the representation of the type and its instances. However, note that triples do not exist physically, rather are they means to logically represent the intermediate representation of data model types and instances. In fact, the physical representation is specified by *reduction rules* which map (sets of) triples to physical objects. The other way round, reduction rules determine how to interpret physical objects in order to extract meaningful information. Based on triple schemes, reduction rules are formulated in an algebraic language.

3 The EAV Algebra

In this section, an informal overview on the EAV algebra is given. A formal definition can be found in the appendix.

The algebra domain consists of two kinds of sets:

1. basic value sets:
   - entity identifiers,
   - attribute names,
   - strings,
   - integers, floats and so on.
2. complex value sets:
   - triples, where a triple consists of an entity identifier, an attribute name, and a value,
   - quadruples, which are similar to triples with an additional index component,
   - tuples,
   - sets,
   - lists,
   - arrays.

In the following sections, we list the corresponding operators for complex value sets. Each description is concluded with a table summarizing the operators and their signatures. As a matter of convention, “any” is a placeholder for an arbitrary type.

Triple and Quadruple Operators

Operators are required to extract parts (i.e. entity, attribute, value, or index components) from triples or quadruples. Thus, four operators are supported to extract the respective components: *ent*, *att, val*, and *ind.*

These operators assume that a specific triple is given, and that simply one component has to be extracted. However, further operators are required to extract e.g. the value of a specific attribute for a given entity. Additionally, there may be multiple triples with equal, entity and attribute names. Hence, *attribute projection* is provided: let *e* be an entity identifier and *a* be an attribute name. Then,

- if applied to a single triple, *attval(e, a)* returns the e’s value for *a*,
- if applied to a list of quadruples, *attval(e, a)* returns a list representing e’s values for a.

- if applied to a set of triples, *attval(e, a)* returns a set representing e's values for a.

In either case, *e.a* is a shorthand for *attval(e, a)*.

Example 4.1
ent ((att, HAS-NAME, att-name)) = att.
attval(att, HAS-NAME) = att-name.
attval(rel, HAS-ATTRTIBUTE) = $<att_1, \ldots, att_n>$.

operator	arguments	result
ent	triple / quadruple	entity identifier
att	triple / quadruple	attribute name
val	triple / quadruple	attribute value
ind	quadruple	index
.	triple x attribute	value
.	set of triples x attribute	set of values
.	set of quadruples x attribute	list of values

Table 1: Triple and Quadruple Operators

Set Operators

The first operators defined for the set value set is *set construction*. "{ }" takes an element as input and creates a set. Furthermore, new sets can be created out of existing ones by means of the operators *set union* ($\cup$) and *set difference* (\). A singleton set can be projected to its element by means of the projection operator *pick*.

Another typical situation requires to transform sets into sets, whereby a given function is applied to each element of the set. This is supported by the *set image* operator. It takes a function (defined on the elements of the set) and a set as input.

Finally, an operator is supported returning the number of elements of the input set (*card*).

name	arguments	result
{ }	any	set of any
$\cup$	set x set	set
\	set x set	set
pick	set of any	any
setimage	function x set	set

Table 2: Set Operators

name	arguments	result
card	set	integer
sort	function x set	list

Table 2: Set Operators

Example 4.2(Set Image)
Let S be the set of sets $\{\emptyset, \{1,4,5\}, \{19, 12, 1962\}\}$.
Then setimage(*s*: *card*(*s*), S) = $\{0, 3\}$.

In this example, *s* is a variable iterating over the input set. In general, functions with one parameter are permitted. However, in an image function one may also refer to variables of enclosing functions. In the sequel, the set of permitted functions is referred to as *function*.

List Operators

Like for sets, a constructor is defined for lists ("< ... >"). Likewise, lists can be appended (*append*). One way to construct a list out of an existing one is to *reverse* the input list. Various operators exist to access parts of a list: a specific element can be identified by its position in the list, the first element can be identified, and the first element can be dropped (*element, first, tail*).

Another useful operator (borrowed from HDBL [Pistor 86]) is a list consisting of all the indexes of a list. For instance, if *l* is a list of three elements, the result of *indexl* is <1, 2, 3>. This operator is useful in order to locate specific elements in a list. One operator (*firsti*) locates the first occurrence of an element in a given list and returns its index, and *alli* returns a list of all indexes where a given element occurs in a list. Essentially, the last two operators are abbreviations (or macros) of *indexl*: e.g., *firsti(e)* can be written as *first(listselect(s: element(l,s)=e, indexl(l))).*

Similar to the case of sets, an image operator exists (*listimage*). Finally, the equivalent to cardinality for sets is *length* for lists.

name	arguments	result
< >	any x .... x any	list
append	list x list	list
reverse	list	list
element	list of any x integer	any
first	list of any	any

Table 3: List Operations

name	arguments	result
tail	list of any	any
indexl	list	list of integer
firsti	list x any	integer
alli	list x any	list of integer
listimage	function x list	list
length	list	integer
sort	function x list	list

Table 3: List Operations

Tuple Operators

In this paragraph we describe the operators specific to tuples. Like with sets, the first operator is a constructor (denoted "[..]"). Note that tuples do not have attribute names, rather are attributes / components identified by indices (this is typical for physical objects). Hence, *comp(t, i)* extracts the i^{th} component of a tuple *t*. Another operator transforms a list into a tuple. Like in the previous two cases of images, a function has to be specified which shall be applied to each element of the list.

name	arguments	result
[ ]	any	tuple
comp	tuple x integer	any
tupleimage	function x list	tuple
indexl	tuple	list of integer
firsti	tuple x any	integer
alli	tuple x any	list of integer

Table 4: Tuple Operators

Example 4.3

Let *E1* be a list of attributes of relation *rel*, i.e.

E_1 = attval(rel, HAS-ATTRIBUTE).

E_1 is a list, since HAS-ATTRIBUTE triples are indexed. Then, construct a tuple of attribute names together with the respective types (E_2).

E_2 = tupleimage(s: [attval(s, HAS-NAME), attval(s, HAS-TYPE)], E_1)

Array Operators

For arrays, only two operators are defined: construction and projection to one element of

name	arguments	result
||		array
element	array x integer	any

Table 5: Array Operators

the array.

Selection

Selection is applicable to sets or lists. Like in the relational algebra, it creates new sets or lists, respectively, consisting of those elements evaluating some specified predicate or formula to *true*.

Before we are able to define selection, selection *predicates* have to be specified. The unary predicates *trip* and *quad* may be applied to any value and are *true* if the argument is a triple or quadruple, respectively.

name	arguments	meaning
trip	any	true if element is triple
quad	any	true if element is quadruple
=	any x any	
≠	any x any	
<	any x any	
>	any x any	
member	any x set	true iff element in set
member	any x list	true iff element in list
⊂	set x set	

Table 6: Selection Predicates

The cardinality operators can be used to abbreviate formulas and conditions in the following way. If sets (lists) of sets (lists) are given, formulas like “all sets (lists), where for exactly (at least, at most) a specific number of element sets a given condition holds” can be formulated. The quantification (exactly, at least, at most) can be checked using the cardinality operators. More formally,

setselect(s: EXACTLY n: f(s), $\{s_1, ..., s_n\}$) = $\{s$ / card(setselect (t: f(t), s)) = n$\}$.
setselect(s: AT MOST n: f(s), $\{s_1, ..., s_n\}$) = $\{s$ / card(setselect (t: f(t), s)) $\leq$ n$\}$.
setselect(s: AT LEAST n: f(s), $\{s_1, ..., s_n\}$) = $\{s$ / card(setselect (t: f(t), s)) $\geq$ n$\}$.

A special case of the "at least" is existential quantification (n = 1), and a special case of "exactly" is universal quantification (n = *card*(s)).

Modification Operators

Up to now, we are able to retrieve or rebuild (sets of) triples and complex values composed out of triples. However, there is no mechanism to "create" triples. In algebras, *construction* is no problem - it simply corresponds to the (recursive) application of constructors to constants. Nevertheless, this is no *creation*, since the constructed value was present in the domain before it was constructed. The semantics of creation in databases actually is to *add* constructed values to some set of values or objects (ie.e. extensions, like relations). Thus, creation and insertion of triples refer to the manipulation of sets of triple schemes (i.e. sets of triples); in that update operators are applied to a triple scheme, specific triples are added and/or removed from a set of triples, thus a set of triples is transformed in a new set of triples. This corresponds to update operations in database systems: their application transforms a database into another, new database. If we assume that only one triple scheme of interest is given at a certain point in time, the triple scheme can be regarded as implicit parameter to update operators.

In the sequel, we shall give the signatures and semantics of update operators, first for triples and then for quadruples.

The creation of a triple just inserts the desired triple into the triple schema.

name	arguments	result
create	entity x string x any	triple scheme
update	entity x string x any	triple scheme
delete	entity x string	triple scheme
update	entity x string x any x integer	triple scheme
delete	entity x string x any x integer	triple scheme

Table 7: Operator for the Modification of Triple Schemes

4 Representation and Realization of Internal Objects

4.1 Triple Schemes

This section introduces the notion of *triple schemes*.

Figuratively speaking, a triple scheme is assigned to a (data model) type or meta type. For (non-meta) types, it reflects information on the structure of the (permitted, possible) instances of the type and its relations to other instances (of possibly distinct types). For meta types, a triple scheme is subdivided into two parts: while one part represents the information of the structure and relations of types (the instances of the meta type), the second one reflects information on the instances of the types (like in the case above). More formally, a triple scheme is a pair, each of which components is a set of triples and/or quadruples.

Comparable to a database schema, a triple schema does not intent to describe *concrete* triples on instance level, rather it specifies *patterns of permissible triples*. Those patterns then are used to describe the structure of schemas, types, and instances conforming to a specific data model. Since we are not interested in concrete instances, *variables* may be used to represent one among a collection of permitted entities or values. Relationships (links, references, and the like) are represented in that the value part of a triple or quadruple contains the entity name of another triple. Beside variables, *constants* are used as a second value of triple components. While a triple scheme actually defines patterns for allowed triples, rules will assume a triple database conforming to the schema.

As an example, a triple schema representing the concepts of the relational model is given in Specification 1.

```
TRIPLE SCHEME relation                                        [1]
      {     <relation, HAS-NAME, rel-name>                    [2]
            <relation, HAS-ATTRIBUTE, att, #> }               [3]
      {     <relation, has-tup, tuple> }                      [4]


TRIPLE SCHEME attribute                                       [5]
      {     <att, HAS_NAME, att-name>                         [6]
            <att, HAS_LEN, att-len> }                         [7]
      {     }                                                 [8]


TRIPLE SCHEME tuple                                           [9]
      {     <tup, HAS_ATTRIBUTE, att, #> }                   [10]
      {     <tup, att-name, val> }                           [11]
```

Specification 1. Relations and Tuples in terms of triple schemes

For triple schemes, some conventions are applied. The third line [3] specifies that relations have attributes. Since the order of attributes is relevant in our example, the set of attributes of one specific relation is represented as a set of quadruples. This is expressed by the “#” as fourth component. Furthermore, variables are written using lower case letters, while constants (attribute names) are written in upper case letters. A variable can refer to another triple (like *att*), as a further convention we assume that variables with the same name always refer to the same entity (i.e. the *tup* in the fourth line refers to [11]). Variables which do not occur as entities elsewhere are assumed to represent basic values (i.e. strings a.s.o.).

Triple schemes more complex than the ones above can get cumbersome to read and understand. Hence, an equivalent ER-like notation is useful. Figure 4 shows a visualization of the schema level of the triple scheme, while Figure 5 shows the instance level. In these figures, boxes represent entities, ovals represent basic values, and arcs represent references.

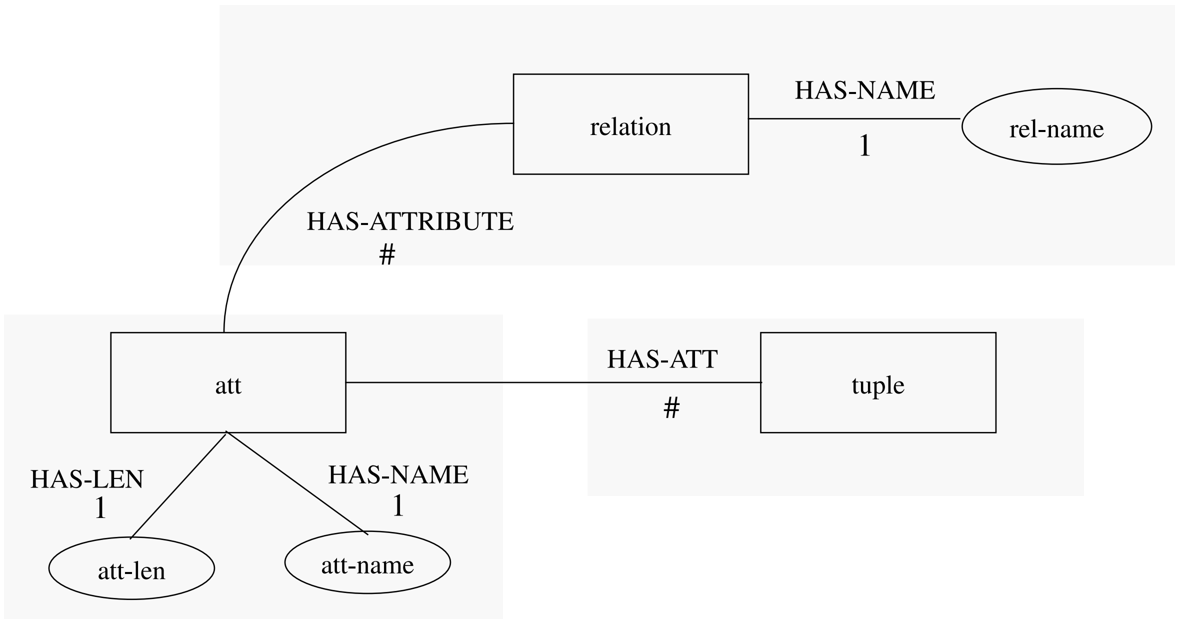


Figure 4 Sample Triple Scheme (Schema Level)

The latter are also labelled with the corresponding attribute and a cardinality. A cardinality is either a constant natural number (e.g. 1), the “#” sign (with the semantics that the arc represents a set of quadruples), or the “*” sign (denoting that multiple triples with the same attribute name can exist, but that those triples do not have to be ordered). Note further, that

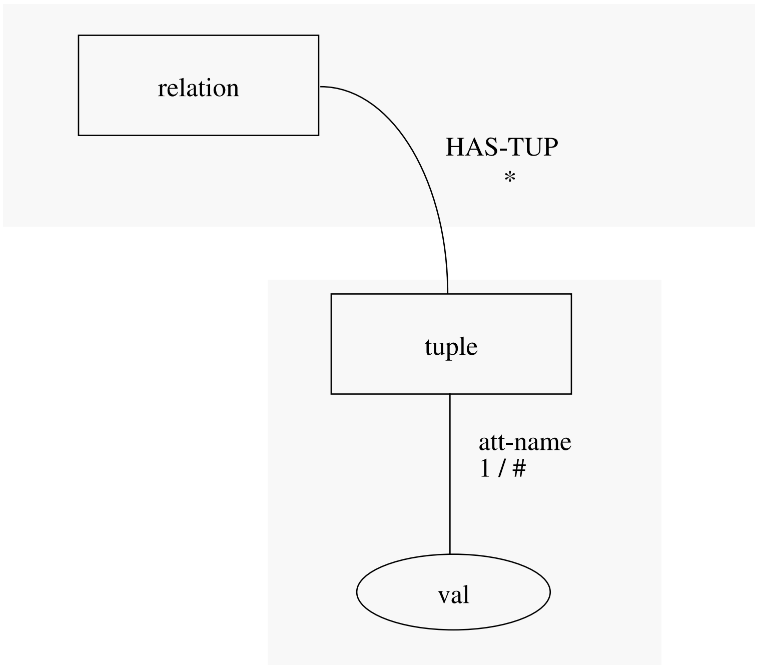


Figure 5 Sample Triple Scheme (Instance Level)

two different kinds of arcs exist (two different kinds of triples): in one case, the attribute (arc label) is a constant identifier (e.g. “HAS-ATT”), while in the other case the attribute is a variable (e.g., *att-name* between *tuple* and *val*). In the latter case, the cardinality is a pair (with both components conforming to one of the three cases above). The reason is that a single cardinality is not sufficient, since it is not clear if e.g. in the case of cardinality 1 there can only be one triple linking a tuple to a value, or whether there can only be one triple for each attribute name. Hence, in this case the first component specifies the cardinality for a concrete attribute, while the second cardinality specifies the attribute-independent cardinality. In the example, there may be only one triple for a specific tuple/attribute pair, but different triples can exist for the same tuple as long as their attribute names are distinct. Therefore, the cardinality is 1/#.

The triple schemes above are rather simple in that they consist of triples/quadruples with always the same entity name. However, it is possible to specify more complex triple schemes with distinct entity names. In this case, it would no longer be clear (the properties of) which entity are to be specified. Therefore, if there are more than one identifiers used in a triple scheme, one of them has to be marked as anchor.

4.2 Reduction

4.2.1 Specification of Reduction

Triple schemes describe the logical representation of objects. This representation is termed “logical” since objects are never stored according to it. In fact, because a triple representation is likely to be space and time inefficient, logical objects are *reduced* to *physical objects*. Here, reduction refers to the fact that the number of triples is much larger than the number of physical objects, and that the number of links or references between triples is much larger than those between physical objects.

Mapping logical objects to physical objects essentially means to transform structured information into unstructured byte sequences. Those unstructured byte sequences are offered by the KIDS object server [Nittel 92]. Hence, reduction rules fulfill two purposes:

- they describe which (parts of) logical objects are mapped together into physical objects,
- they describe the format to be superimposed on physical objects at retrieval.

For reduction, it is assumed that operations of the algebra are implemented on top of the object server, and that they realize the structures described above (i.e., tuples, sets, lists, and arrays).

Now, a reduction rule is an algebraic expression and is attached to a triple scheme. Each reduction rule has the semantics to construct a physical object whenever it is applied. A triple scheme, on the other hand, may have multiple reduction rules attached to it. If we understand a triple scheme (or, more precisely, the instance part of a triple scheme) as one logical object, having multiple reduction rules refers to fragmentation and/or replication, since multiple physical objects are created from one logical one.

In the example above, the corresponding reduction rules could be specified as follows.

```
REDUCTION relation                                                  [12]
        { rel.HAS_ATTRIBUTE }                                       [13]
        { rel.HAS-TUPLE}                                            [14]

REDUCTION attribute                                                 [15]
        { [ att_name, att_len] }                                    [16]

REDUCTION tuple                                                     [17]
        { tup.HAS_ATTRIBUTE }                                       [18]
        { tupleimage( s: tup.s, tup.HAS_ATTRIBUTE.HAS-NAME ) }      [19]
```

Specification 2. Sample Reduction Rules

4.2.2 Implementation of Reduction

So far, we are able to specify triple schemes and reduction rules. One still open point is how to realize reduction, i.e., the approach how to generate code that maps triple schemes to physical objects or to any intermediate level. To that end, three possibilities can be distinguished:

- we assume a *triple machine* that directly realizes triples and quadruples,
- triple schemes are directly realized in terms of the EAV algebra,
- triple schemes are mapped to physical objects.

As long as no triple hardware is available, the first alternative will lead to a high degree of inefficiency. It will, however, we useful for a prototype, and as an intermediate solution until a full-fledged object server is available.

In the second possibility, we assume an intermediate layer that is defined by the EAV algebra. Hence, the interface of this layer would realize complex values and would be responsible to map these values to physical objects. Advantages of this approach may be a better degree of extensibility and the potential use of complex values and their operations for purposes like query optimization.

Finally, following the last approach, triple schemes and reduction rules are directly mapped to physical objects and operations on those. For the chosen approach and details, see [Scherrer 93].

5 Open Issues and Discussion

Beside the description of a meta data model and its implementation (which can be found elsewhere [Scherrer 93]), some major points are yet open in the context of EAV.

First, we have assumed so far that the object manager is realized as one cohesive block of software, with little further modularization. Additionally, we assumed that the object manager is realized by means of generation, such that it can be impossible to add further subcomponents without modification to the object manager. Furthermore, it is still an unanswered question how EAV is affected by extensions and modifications. For instance, assume multi-version concurrency control to be applied by the transaction management subsystem. In this case, while the concurrency control would have to be realized by the transaction management, the object manager has to implement version management (e.g., has to support the creation of new object versions). While the data model level is not affected (since versions are used for internal purposes only), the internal realization of the corresponding object management subsystem will be different (and will have to be generated such that it supports the additional functionality). Hence, for this part of DBMS construction, it may also be necessary to support various realization strategies (instead of one single, uniform mapping).

Second, a related problem refers to extending the object manager with additional functionality in the sense of adding further subcomponents. For example, if the data manipulation language offers declarative, collection-oriented access to objects, we typically will have to support algebraic and non-algebraic optimization. Hereby, it seems straightforward to use the EAV algebra for non-algebraic optimization. However, it is still unclear how various optimizers can be included in the object manager, and how a generated optimizer [Graefe 87] can be integrated with the rest of object management (which is also generated).

A final open problem is extensibility in EAV, whereby extensibility of EAV refers to the definition of new value sets and corresponding operators. For instance, in advanced applications like multi-media applications or geographic information systems, neither long fields nor the complex values supported so far might be sufficient. Hence, it may be desired to specify and implement new value sets such that they can be used in those cases where the original algebra is not sufficient.

As for related work, the EAV algebra is similar to other algebras supporting complex values or complex objects with orthogonal value constructors [Abiteboul 88, Geppert 90, Vandenberg 91]. They differ from EAV, however, in the concrete collection of value constructors and in that they do not support triples and quadruples. Furthermore, triple schemes can be graphically represented by (extended) ER-diagrams [Chen 76]; hence, both EAV and ER are quite similar. Finally, EAV is comparable to the Telos model [Mylopoulos 90], while Telos is based on logic and offers higher expressive power.

6 Summary

In this paper, we have introduced the EAV model. EAV supports the definition of triple schemes, consisting of (patterns of permissible) triples and quadruples. In our context (the KIDS project) these schemes are used for the conceptual representation of data model structures. Furthermore, we have described reduction rules as means to map conceptual, internal objects to complex values or physical objects. Finally, we addressed open issues to be dealt with in order to reasonably integrate EAV in the construction process of DBMS.

Both, a demonstration of EAV for a complex data model (NO^2) as well as investigations on the use of EAV for transaction management can be found elsewhere [Knaus 91, Schramm 92].

7 References

Abiteboul 88 S. Abiteboul, C. Beeri: *On the Power of Languages for the Manipulation of Complex Objects*. Technical Report 846, INRIA.

Chen 76 P.P. Chen: *The Entity-Relationship Model - Towards a Unified View of Data*. ToDS 1:1, 1976.

Dittrich 91 K.R. Dittrich, A. Geppert, V. Goebel, S. Nittel, S. Scherrer: *KIDS: A Declarative Approach to DBMS Construction*. In S. Spaccapietra (Ed.): Proc. SI-Conference on Database Research in Switzerland, Lausanne, 1991.

Geppert 90 A. Geppert, K.R. Dittrich, V. Goebel, S. Scherrer: *An Algebra for the NO^2 Data Model*. Project Report ITHACA.Unizh.X.4#3, University of Zuerich.

Geppert 92a A. Geppert: *On the Architecture of Generated DBMS and DBMS Generators Or: Can Kids Generate DBMS?* Technical Report, Institut für Informatik, Universität Zürich, May 1992.

Geppert 92b A. Geppert, S. Scherrer: *On KIDS and Concepts: Requirements for a Data Model Specification Language*. Technical Report, Institut für Informatik, Universität Zürich, 1992.

Graefe 87 G. Graefe, D.J. DeWitt: *The EXODUS Optimizer Generator*. Proc. SIGMOD 1987.

Knaus 91 M. Knaus: Transformation of a structurally object-oriented Data Model into a Tripel Kernel Database System. Master Thesis, Institut für Informatik, Universität Zürich, March 1991 (in German).

Mylopoulos 90 J. Mylopoulos, A. Borgida, M. Jarke, M. Koubarakis: *Telos: Representing Knowledge About Information Systems*. ACM ToIS 8:4, 1990.

Nittel 92 S. Nittel: *The KIDS Object Server Interface*. Technical Report, Institut für Informatik, Universität Züerich, 1992.

Pistor 86 P. Pistor, F. Andersen: *A Database Language for Sets, Lists and Tables*. Information Systems 11:4, 1986.

Scherrer 93 S. Scherrer: *The KIDS Data Model Specification Language*. Technical Report 93.14, Institut für Informatik, Universität Zürich, April 1993.

Schramm 92 M. Schramm: *Specification of Synchronisation Techniques in Cnfigurable Database Systems.* Master Thesis, Fakultät für Informatik, Universität Karlsruhe, 1992 (in German).

Vandenberg 91 S.L. Vandenberg, D.J. DeWitt: *Algebraic Support for Complex Objects with Arrays, Identity, and Inheritance*. Proc. SIGMOD 1991.

A The EAV Algebra

This section contains a formal definition of the EAV algebra. We start with the definition of basic value sets (*sorts)* and later on describe complex value sets.

Basic Value Sets

For that matter, define the following value sets:

Definition A.1 (Characters)
Let $\mathcal{C}$ be the set of printable characters.

Definition A.2 (Strings)
$\mathcal{S} = \cup_i \mathcal{C}^i$ be the set of strings, where s is an element of $\mathcal{C}^i$ iff there are $c_1, ..., c_i \in \mathcal{C}$ such that $s = c_1c_2...c_i$.

Definition A.3 (Entities)
$\mathcal{E} \subseteq \mathcal{S}$ is the set of *entity names.*

Definition A.4 (Attribute Names)
$\mathcal{A} \subseteq \mathcal{S}$ is the set of *attribute names*.

Definition A.5 (Boolean)
$\mathcal{Bool} = \{\text{true, false}\}$ is the set of boolean values.

Definition A.6 (Integers)
$\mathcal{I}$ is the set of integers.

Definition A.7 (Floats)
$\mathcal{F}$ is the set of floating point numbers.

Definition A.8 (Byte Sequences)
$\mathcal{B}$ is the set of byte sequences.

Furthermore, we assume that values are elements of one of the basic value sets, including entity names[4].

Definition A.9 (Basic Values)
$\mathcal{BV} = \mathcal{C} \cup \mathcal{E} \cup \mathcal{S} \cup \mathcal{A} \cup \mathcal{Bool} \cup \mathcal{I} \cup \mathcal{F} \cup \mathcal{B}$ is the set of valid *basic values*.

Definition A.10 (Values)
$\mathcal{V}$ is the union of all valid (basic or complex) value sets.

A.1 Triples and Quadruples

Definition A.11 (Triples)
An EAV-triple is a tuple (e,a,v), where $e \in \mathcal{E}$, $a \in \mathcal{A}$, and $v \in \mathcal{BV}$.
$\mathcal{T} = \mathcal{E} \times \mathcal{A} \times \mathcal{BV}$ is the set of all valid triples.

Definition A.12 (Quadrupels)
An EAV-quadruple is a tuple (e,a,v,i), where where $e \in \mathcal{E}$, $a \in \mathcal{A}$, $v \in \mathcal{BV}$, and

4. For the sake of conciseness, we do not distinguish the various basic value sets.

$i \in \mathcal{N}$, where $\mathcal{N}$ is the set of natural numbers.
$\mathcal{Q} = \mathcal{E} \times \mathcal{A} \times \mathcal{BV} \times \mathcal{N}$ is the set of all valid quadruples.

Definition A.13 (Database)
$\mathcal{D} \subseteq \mathcal{T} \cup \mathcal{Q}$ is a database.

Definition A.14 (Triple Projection)
ent : $T \cup \mathcal{Q} \rightarrow \mathcal{E}$
ent ((e,a,v)) = e,
ent((e,a,v,i)) = e.

att: $\mathcal{T} \cup \mathcal{Q} \rightarrow \mathcal{A}$
att ((e,a,v)) = a,
att((e,a,v,i)) = a.

val: $\mathcal{T} \cup \mathcal{Q} \rightarrow \mathcal{BV}$
val ((e,a,v)) =v,
val((e,a,v,i)) = v.

ind: $\mathcal{Q} \rightarrow I$
ind((e,a,v,i)) = i.

Definition A.15 (Triple Projection)
attval : $\mathcal{E} \times \mathcal{A} \rightarrow \mathcal{BV}$.
attval(e, a) = v, if there exists exactly one v, such that $(e, a, v) \in \mathcal{D}$. $E.a$ is a shorthand for attval(e, a).

attval : $\mathcal{E} \times \mathcal{A} \rightarrow \mathcal{List}$.
attval(e, a) = $<v_1,...,v_n>$, if there are quadruples $<e,a,v_1, i_1>$, ..., $<e,a,v_n, i_n>$ $\in \mathcal{D}$ such that $i_j < i_k$ for $1 \leq j < k \leq n$. $E.a$ is a shorthand for attval(e, a).

Definition A.16 (Creation for Triples)
create: set($\mathcal{T} \cup \mathcal{Q}$) $\times \mathcal{E} \times \mathcal{A} \times \mathcal{BV} \rightarrow$ set($\mathcal{T} \cup \mathcal{Q}$)
create(S, e, a, v) = S $\cup$ { (e,a,v) }.

Definition A.17 (Update for Triples)
update: set($\mathcal{T} \cup \mathcal{Q}$) $\times \mathcal{E} \times \mathcal{A} \times \mathcal{BV} \rightarrow$ set($\mathcal{T} \cup \mathcal{Q}$)
update(S, e, a, v) = S \ setselect(s: ent(s) = e $\wedge$ att(s) = a, S) $\cup$ { (e,a,v) }.

Definition A.18 (Delete for Triples)
delete: set($\mathcal{T} \cup \mathcal{Q}$) $\times \mathcal{E} \times \mathcal{A} \rightarrow$ set($\mathcal{T} \cup \mathcal{Q}$)
delete(S, e, a) = S \ setselect(s: ent(s) = e $\wedge$ att(s) = a, S).

Definition A.19 (Creation for Quadruples)
create: set($\mathcal{T} \cup \mathcal{Q}$) $\times \mathcal{E} \times \mathcal{A} \times \mathcal{BV} \rightarrow$ set($\mathcal{T} \cup \mathcal{Q}$)
create(S, e, a, v) = S $\cup$ { (e,a,v, i) }, where i > max { j / $\exists$ w: (e,a,w,j) $\in$ S}.

A.2 Functions

Definition A.20 (Functions)
$\mathcal{Function}$ is the set of functions with domain $\mathcal{V}$ and range $\mathcal{V}$.
f $\in$ $\mathcal{Function}$ is denoted *s: f(s)*, were f is a parameterized algebraic expression and *s* is a variable.

A.3 Predicates and Formulas

Definition A.21 (Unary Predicates)
trip : $\mathcal{V} \rightarrow \mathcal{Bool}$
quad: $\mathcal{V} \rightarrow \mathcal{Bool}$.

Definition A.22 (Binary Predicates)
op: $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{Bool}$, where op $\in$ $\{=, \neq, <, >, \in, member, \subset\}$.

Definition A.23 (Predicates)
$\mathcal{P}$ is the set of (unary and binary) predicates.

Definition A.24 (Formulas)
If b $\in$ $\mathcal{Bool}$, then b is a formula.
Iff b_1, b_2 are formulas, then the following are well-formed formulas as well:
- $b_1 \wedge b_2$
- $b_1 \vee b_2$
- $\neg\, b_1$
- $\forall$x: b1.

$\mathcal{Formula} \subset \mathcal{Function}$ is the subset of formulas with range $\mathcal{Bool}$.

A.4 Sets

Definition A.25 (Sets)
$\mathcal{Set}$ is a complex value set. {} (the empty set) is a member of $\mathcal{Set}$ as well as $\{v_1, ..., v_n\}$ for $v_i \in \mathcal{V}$.

Definition A.26 (Set Constructors)
$\{\ \} : \mathcal{V} \rightarrow \mathcal{Set}$.

Definition A.27 (Set Theoretic Operators)
$\cup : \mathcal{Set} \times \mathcal{Set} \rightarrow \mathcal{Set}$.
$\backslash : \mathcal{Set} \times \mathcal{Set} \rightarrow \mathcal{Set}$.

Definition A.28 (Set Projection)
pick: $\mathcal{Set} \rightarrow \mathcal{V}$.
pick ({e}) = e.

Definition A.29 (Set Image)
setimage : $\mathcal{Function} \times \mathcal{Set} \rightarrow \mathcal{Set}$.
setimage $(s{:}\, f(s), v) = \{\ x / \exists\ y \in v \wedge x = f(y)\}$.

Definition A.30 (Cardinality)
$card : \mathcal{Set} \rightarrow \mathcal{N}$
$card(\ \{v_1,..., v_n\}\ ) = n.$

Definition A.31 (Set Selection)
setselect : $\mathcal{Formula} \times \mathcal{Set} \rightarrow \mathcal{Set}$.
setselect (s: f(s), v) = $\{x \,/\, x \in v \wedge f(x)\ \}$.

Definition A.32 (Sorting)
setsort : $\mathcal{Function} \times \mathcal{Set} \rightarrow \mathcal{List}$
setsort (s: f(s), ($\{v_1,..., v_n\}$) = $\langle v_{i1}, ..., v_{in} \rangle$,
where $\{i_j \,/\, 1 \le j \le n\ \} = \{1, .., n\}$ and
$f(\ v_{ij}\ ) \le f(\ v_{ik}\ )$ for $j \le k$.

A.5 Lists

Definition A.33 (Lists)
$\mathcal{List}$ is a complex value set. $\langle\rangle$ (the empty list, *NIL*) is a member of $\mathcal{List}$ as well as $\langle v_1, ...,v_n \rangle$ for $v_i \in \mathcal{V}$.

Definition A.34 (List Constructor)
$\langle\rangle$: $V^n \rightarrow \mathcal{List}$.

Definition A.35 (Append)
append : $\mathcal{List} \times \mathcal{List} \rightarrow \mathcal{List}$.
append (v , $\langle\rangle$) = append($\langle\rangle$, v) = v.
append($\langle v_1, ..., v_n \rangle, \langle w_1, ..., w_m \rangle) = \langle v_1, ..., v_n, w_1, ..., w_m \rangle$.

Definition A.36 (Reverse)
reverse: $\mathcal{List} \rightarrow \mathcal{List}$
reverse($\langle v_1, ..., v_n \rangle) = \langle v_n, ..., v_1 \rangle$

Definition A.37 (List Projection)
element : $\mathcal{List} \times I \rightarrow \mathcal{V}$.
element ($\langle v_1, \ldots, v_n \rangle$, i)= v_i.

first : $\mathcal{List} \rightarrow \mathcal{V}$.
first ($\langle v_1, \ldots, v_n \rangle$) = v_1.

tail : $\mathcal{List} \rightarrow \mathcal{List}$.
tail ($\langle v_1, \ldots, v_n \rangle$) = $\langle v_2, \ldots, v_n \rangle$.

Definition A.38 (Index List)
indexl: $\mathcal{List} \rightarrow \mathcal{List}$.
indexl(nil) = nil,
indexl($\langle v_1, ..., v_n \rangle) = \langle 1, ..., n \rangle$.

Definition A.39 (First Index)
firsti: $\mathcal{List} \times \mathcal{V} \rightarrow \mathcal{I}$.
firsti (l, v) = first(listselect(s: element(l, s)=v, indexl(l)))

Definition A.40 (All Indexes)
firsti: $\mathcal{List} \times \mathcal{V} \rightarrow \mathcal{List}$
alli (l, v) = listselect(s: element(l, s)=v, indexl(l))

Definition A.41 (List Image)
listimage: $\mathcal{Function} \times \mathcal{List} \rightarrow \mathcal{List}$
listimage (s: f(s), $<>$) = $<>$.
listimage (s: f(s), $<v_1, \ldots, v_n>$) = $<f(v_1), \ldots f(v_n)>$.

Definition A.42 (Length)
length: $\mathcal{List} \rightarrow \mathcal{I}$
length ($<v_1, \ldots, v_n>$) = n.

Definition A.43 (Selection for Lists)
listselect: $\mathcal{Formula} \times \mathcal{List} \rightarrow \mathcal{List}$
listselect (s: p(s), $<>$) = $<>$,
listselect (s: p(s), $<v_1, \ldots, v_n>$) =
append($<v_1>$, listselect(s: p(s), tail($<v_1, \ldots, v_n>$))), if $p(v_1)$,
σ_L [$\lambda s.\, p$] ($v_2, \ldots, v_n>$), otherwise.

Definition A.44 (Sorting)
sort : $\mathcal{Function} \times \mathcal{List} \rightarrow list$
setsort (s: f(s), $<v_1, \ldots, v_n>$) = $<v_{i1}, \ldots, v_{in}>$,
where $\{i_j \,/\, 1 \leq j \leq n\} = \{1, .., n\}$ and
$f(v_{ij}) \leq f(v_{ik})$ for $j \leq k$.

A.6 Tuples

Definition A.45 (Tuple Constructor)
[ ] : $\mathcal{V}^n \rightarrow \mathcal{Tuple}$.

Definition A.46 (Tuple Projection)
comp: $\mathcal{Tuple} \times \mathrm{I} \rightarrow \mathcal{V}$
comp ($[v_1, \ldots, v_n]$, i) = v_i, if $1 \leq i \leq n$ and where v_i in $\mathcal{V}$.

Definition A.47 (Tuple Image)
tupleimage : $\mathcal{Function} \times \mathcal{List} \rightarrow \mathcal{Tuple}$
tupleimage(s: f(s), $<v_1, \ldots, v_n>$) = $[f(v_1), \ldots f(v_n)]$.

Definition A.48 (Index List)
indexl: $\mathcal{Tuple} \rightarrow \mathcal{List}$.
indexl($[v_1, \ldots, v_n]$) = $<1, \ldots, n>$.

Definition A.49 (First Index)
firsti: $\mathcal{Tuple} \times \mathcal{V} \rightarrow I$.
firsti (t, v) = first(listselect(s: comp(t, s)=v, indexl(t)))

Definition A.50 (All Indexes)
firsti: $\mathcal{Tuple} \times \mathcal{V} \rightarrow \mathcal{List}$
alli (t, v) = listselect(s: element(t, s)=v, indexl(t))

A.7 Arrays

Definition A.51 (Arrays)
$\mathcal{Array}$[n] is a complex value set. If $v_1, \ldots, v_n$ are elements of $\mathcal{V}$, then $|v_1, \ldots, v_n|$ is a member of $\mathcal{Array}$[n].

Definition A.52 (Array Constructor)
$|\,|: \mathcal{V}^n \rightarrow \mathcal{Array}[n]$.

Definition A.53 (Array Projection)
$\pi_A : I \times \mathcal{Array}[n] \rightarrow \mathcal{V}$.
$\pi_A (i, [v_1, \ldots, v_n]) = v_i$, if $1 \le i \le n$.